Implementation funktionaler Programmiersprachen durch Quelltexttransformation

Dissertation

zur Erlangung des akademischen Grades

Doctor rerum naturalium (Dr. rer. nat.)

im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät II

der Humboldt-Universität zu Berlin

von

Dipl.-Math. Dragan Maćoš

geboren am 18. April 1967 in Zrenjanin, Jugoslawien

Präsident der Humboldt-Universität zu Berlin
Prof. Dr. Dr. h. c. Hans Meyer

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II
Prof. Dr. Wolfgang Reisig

Gutachter:

1. Prof. Dr. Klaus Bothe

2. Prof. Dr. Elfriede Fehr

3. Prof. Dr. Hans-Dieter Burkhard

Tag der mündlichen Prüfung: 03.7.98

Implementation funktionaler Programmiersprachen durch Quelltexttransformation

Dragan Maćoš

Institut für Informatik
der Humboldt-Universität zu Berlin

III

Zusammenfassung

Die vorliegende Arbeit soll einen Beitrag zur Entwicklung von Compilern funktionaler Sprachen zur Erzeugung von Zielcode in einer höheren prozeduralen Programmiersprache liefern. In der Dissertation werden mehrere Implementationstechniken funktionaler Sprachen analysiert, wobei für jede Technik ein Verfahren zur Realisierung eines auf der Übersetzung in eine prozedurale Sprache basierenden Compilers angegeben wird. Durch die verschiedenen Implementationsverfahren, die in der Arbeit analysiert bzw. definiert wurden, sind die beiden großen Klassen funktionaler Sprachen abgedeckt worden: strikte und nicht-strikte funktionale Programmiersprachen.

Die Dissertation kann in drei größere Teile gegliedert werden, die sich mit folgenden Bereichen beschäftigen:

1. Direkte Übersetzungen funktionaler in prozedurale Programmiersprachen
2. Übersetzung des Codes der abstrakten *SECD*-Maschine in eine prozedurale Sprache
3. Erzeugung des Zielcodes in einer prozeduralen Programmiersprache bei Graph-Reduktion-basierten Implementationstechniken.

Im ersten Teil, dem Schwerpunkt der Arbeit, wird eine über existierende Ansätze hinausgehende Transformation einer einfachen strikten funktionalen Programmiersprache (erweiterter Lambda-Kalkül) in den Code einer prozeduralen Zielsprache eingeführt und durch ein formales System von Transformationsregeln beschrieben. Die Transformation zeichnet sich durch Transparenz und Einfachheit (Erzeugung minimalen Codes) aus, wodurch ein darauf aufbauender von uns realisierter Compiler im Bezug auf andere vergleichbare Implementationen erhebliche Laufzeitvorteile besitzt.

Im zweiten und dritten Teil werden die aus der Literatur bekannten Implementationstechniken modifiziert bzw. optimiert, um Zielprogramme in einer höheren prozeduralen Programmiersprache erzeugen zu können. Für die Graph-Reduktion-basierte Technik haben wir eine Prototypimplementation vorgenommen.

Abstract

This work contributes methods for the design and implementation of translation schemes for functional programming languages with procedural programming languages as a target. The dissertation analyzes different implementation techniques of functional languages. For each technique an approach is defined for the implementation of a compiler that translates the functional source language into a procedural target language. The analyzed techniques cover both classes of functional languages, strict and non-strict ones. The work is structured into three parts with the following topics:

1. direct translation of functional programming languages into procedural programming languages;
2. translation of the code of the SECD-abstract machine into a procedural language;
3. emission of a target code in a procedural language by graph-reduction-based implementation techniques.

In the first part, we give a formal definition for translating a simple functional language (syntactic sugared lambda calculus) into the code of a procedural programming language. The defined transformation is simple and transparent.

In the second and the third part, existing translation schemes are modified, i. e. two known implementation techniques are optimized to emit target code in an procedural language.

Danksagung

Die vorliegende Dissertation ist von 1995-1997 während meiner vom Senat des Landes Berlin durch ein NaFöG-Stipendium finanzierten Tätigkeit an der Humboldt-Universität zu Berlin entstanden.

Ich möchte besonders meinem Betreuer, Herrn Prof. Dr. Klaus Bothe, danken. Durch seine fruchtbaren Anregungen und Diskussionen prägte er meinen wissenschaftlichen Werdegang. Die vorliegende Arbeit hat er aufmerksam mehrmals gelesen und jedesmal eine wertvolle Kritik geliefert. Die ganze Zeit unserer Zusammenarbeit war er ein aufmerksamer und immer hilfsbereiter Freund.

Mein besonderer Dank gilt auch Frau Prof. Dr. Elfriede Fehr von der Freien Universität Berlin. Neben ihrer Bereitschaft, das Zweitgutachten für die vorliegende Dissertation zu übernehmen, hat sie mit wichtigen Hinweisen zur Fertigstellung der Endversion der Dissertationsschrift beigetragen.

Desweiteren danke ich Herrn Prof. Dr. Hans-Dieter Burkard für sein Interesse an meiner Arbeit und konstruktive Diskussionen, die wir am Institut für Informatik führten.

Allen meinen Kollegen und Freunden an der Lehr- und Forschungseinheit ,,Softwaretechnik und Theorie der Programmierung II" am Institut möchte ich für die angenehme und motivierende Arbeitsatmosphäre danken. Besonderer Dank gehört meinem Kollegen und Freund Dr. Frank Müller, mit dem ich auch einen großen Teil meiner Freizeit verbracht habe. Ich bedanke mich bei Nikolaos S. Papaspyrou von der National Technical University of Athens für die während einiger Phasen meiner Arbeit geführte wertvolle Diskussion. Besonders hilfsreich waren für mich auch die Kommentare von Torben Ægidius Mogensen von der Universität Copenhagen. Desweiteren bedanke ich mich bei Manuel M. T. Chakravarty von der Universität Tsukuba für seine Bereitschaft, mir beim Lösen einiger Probleme zu helfen.

Meine Eltern haben mir das Studium ermöglicht. Durch ihre Erziehung wurde in mir eine besondere Motivation für eine wissenschaftliche Tätigkeit geweckt. Meinen Freunden, die ich in Deutschland kennengelernt habe, danke ich für die Hilfe meine ,,Downs" zu überwinden und das Glück meiner ,,Ups" mit mir zu teilen. Außerdem möchte ich mich besonders bei meinem Freund Zoran Barać bedanken.

Meine schwierigsten und glücklichsten Momente habe ich mit meiner Freundin Nicola Schrenk verbracht. Ihre Liebe und Aufmerksamkeit waren eine wichtige Hilfe auf dem Weg zu dieser Dissertation.

Inhaltsverzeichnis

Kapitel 1

Einleitung und Motivation

1.1 Grundlagen, Vorzüge und Probleme funktionaler Programmierung

Die Zielstellung der vorliegenden Dissertation besteht darin, die Vorteile funktionaler Programmierung mit den Vorteilen prozeduraler Programmierung zu verbinden. Dazu betrachten wir die Transformation zwischen beiden Programmierstilen bzw. Programmierparadigmen als eine Implementationstechnik funktionaler Sprachen.

Ebenso wie die logischen Programmiersprachen werden die funktionalen Sprachen den deskriptiven (beschreibenden) Sprachen zugeordnet. Im Gegensatz zu prozeduralen (imperativen) Sprachen steht nicht ein Algorithmus im Mittelpunkt, der ein zu behandelndes Problem löst. Vielmehr wird versucht, Programmierung auf einem abstrakten Niveau zu betreiben, bei dem ein Programm der Problembeschreibung näher kommt und Teile der Problemlösung (Algorithmen) der Sprachimplementation überlassen werden. So stützt sich die logische Programmierung auf den Apparat der mathematischen Logik. Im funktionalen Paradigma werden Funktionen im mathematischen Sinne verwendet.

Wenn wir es hier auch mit einer etwas idealisierten Sicht zu tun haben, kommt der deskriptive Ansatz in vielen Fällen diesem Ideal nahe, während jedoch für bestimmte Anwendungsfälle Erweiterungen deskriptiver Sprachen im Sinne der prozeduralen Programmierung vorgenommen werden (insbesondere die Erweiterung um den Zustandsbegriff - globale Variablen).

Funktionale Programmierung zeichnet sich von anderen deskriptiven Sprachen dadurch aus, daß grundsätzlich Funktionen zur Problembeschreibung verwendet werden. Dabei kann im Sinne von Bird und Wadler [BW92] eine Präzisierung des Begriffs vorgenommen werden:

> *Die funktionale Programmierung ist ein Programmierungsstil, wobei es im wesentlichen darum geht, Definitionen zu erzeugen und Ausdrücke mit Hilfe eines Computers auszuwerten.*

Ein funktionales Programm besteht also aus Ausdrücken und Definitionen. Während der Berechnung der Ausdrücke werden die Definitionen angewendet. Die Berechnung eines funktionalen Programms läuft wie in einem Taschenrechner ab: Ein Ausdruck wird ausgewertet, indem schon definierte Funktionen aufgerufen werden[1].

Ein (kurzes) Beispiel, die Berechnung der Fakultät einer natürlichen Zahl, soll diese Definition illustrieren:

```
(define fakt
  (lambda (n)
    (if (= n 1)
      1
      (* n (fakt (- n 1))))))
```

In diesem Beispiel wird die Fakultätsfunktion **fakt** in der funktionalen Sprache *Scheme* [RE91] definiert. Die Definition ist rekursiv, wobei die Rekursion eine der wichtigsten Methoden der funktionalen Programmierung ist. Ein Ausdruck der Form

(fakt 2)

wird dann durch den Computer ausgewertet, indem obige Definition angewendet wird. Aus (fakt 2) wird schrittweise:

(* 2 (fakt (- 2 1)))
(* 2 (fakt 1))
(* 2 1)
2

Eine weitere Begriffsbestimmung der funktionalen Programmierung findet sich bei [FG93]. Hier wird eine äußerst knappe, konzentrierte Begriffsbildung vorgenommen:

> *Eine funktionale Sprache ist ein System zur Bezeichnung der Bedeutung primitiver Werte und Funktionen.*

Der oben erwähnte allgemeine Vorteil funktionaler Sprachen im Sinne einer deskriptiven Herangehensweise kann nun weiter präzisiert werden. In der Literatur

[1] „Functional programming is based on the idea of calculation. We define ourselves the functions to be used, and the implementation will calculate the values of expressions which use these functions, just as a traditional calculator will calculate arithmetical expressions like those given above" [Tho95], Seite 6.

[WC93, Wat90, FG93, Gol94] werden u. a. folgende wesentliche Vorzüge funktionaler Programmierung genannt:

- schnellere Softwareentwicklung
- kürzere Programme
- lesbarer Code
- leichtere Verifikation des Codes
- besser geeignet für die parallele Evaluation
- deklarative Beschreibung der Lösung eines Problems
- mathematische Grundlage (lambda-Kalkül und denotationale Semantik)

Die hier aufgeführten Vorzüge resultieren aus einer Reihe spezieller Eigenschaften und Konzepte funktionaler Sprachen:

- Klarheit der Abarbeitung: Substitutionsprinzip der Ausdrücke und ihrer Werte (referentielle Transparenz)
- der funktionale Programmierungsstil ist kein sequentionelles Berechnungsmodell (sondern implizit parallel)
- flexible (oft statische) Typsysteme
- mathematische Notation höherer Ordnung (z. B. Funktionen als Werte)
- viele funktionale Sprachen haben eine nicht-strikte Semantik (welche die Abarbeitung unendlicher Datenstrukturen unterstützt).

Trotz der hier genannten Vorzüge funktionaler Programmierung ist das prozedurale Paradigma das mit Abstand am weitesten verbreitete. Die wichtigsten Probleme für eine breitere Anwendung funktionaler Programmiersprachen sind die folgenden:

- Zustände und ihre Änderungen (welche rein funktionale Programmiersprachen nicht unterstützen) erscheinen in der realen Welt, so daß die Probleme dieser Art schwerer zu beschreiben sind.
- Bei der Anwendung funktionaler Programmierung ist mathematisch geschultes Denken vorteilhaft (das fehlt oftmals in der Ausbildung der Programmierer).

- Die Abarbeitung funktionaler Programme hat oft eine geringere Effektivität als die Abarbeitung prozeduraler Programme.
  Das Problem ist heute nicht mehr in dem Maße aktuell wie früher. Es gibt heute Implementationen funktionaler Programmiersprachen, die der Effektivität nach sogar besser als bekannte Implementationen einiger prozeduraler Sprachen sind.

Im Gegensatz zur funktionalen Programmierung zeichnet sich die prozedurale (imperative) dadurch aus, daß der Begriff des Algorithmus im Mittelpunkt steht: Algorithmen verändern bei ihrer Arbeit die Werte von Variablen.

Das oben angegebene funktionale Programm zur Berechnung der Fakultät sieht prozedural (mit einem Zyklus) so aus:

```
PROCEDURE Fakt(n: INTEGER): INTEGER;
VAR
  Res: INTEGER;
BEGIN
  Res := 1;
  WHILE (n > 0)  DO
    Res := Res*n;
    n := n-1
  END;
  RERTURN Res
END Fakt;
```

Damit ist der Schritt von der Problemlösung zur Implementation vollzogen, die auf einem niederen Abstraktionsniveau liegt. Prozedurale Programmiersprachen sind weitverbreitet, und ihre Compiler sind oftmals hocheffektiv implementiert.

Ein naheliegender Ansatz besteht damit darin, auf höherem Abstraktionsniveau funktional zu programmieren und das (fertige) Programm dann in ein prozedurales zu übersetzen, das dann mit einem optimierenden Compiler weiterverarbeitet werden kann. Ggf. sind auch manuelle Eingriffe in ein automatisch erzeugtes prozedurales Programm denkbar.

Auf vorhandene Ansätze, Vorteile bzw. Ziele einer derartigen Transformation zwischen funktionalen und prozeduralen Sprachen werden wir später (Abschnitte 1.4 und 1.5) näher eingehen.

1.2 Konzepte und Sprachen für die funktionale Programmierung

Wenn wir uns der Implementation funktionaler Sprachen zuwenden, in unserem Fall durch Transformation in prozedurale Sprachen, so müssen wir von einer genaueren

Analyse ausgehen. Die tatsächlich verwendeten funktionalen Sprachen sind vielfältig, ebenso die einbezogenen Konzepte.

Neben dem Grundprinzip, Funktionen zu verwenden, kommt eine Reihe wichtiger weiterführender Konzepte zur Anwendung: Funktionen höherer Ordnung, Pattern Matching, Auswertungsstrategien, Polimorphismus, Datentypen usw. Mit der Entwicklung der funktionalen Programmiersprachen wurden auch diese Konzepte schrittweise eingeführt bzw. modifiziert. Wir wollen an dieser Stelle eine knappe Übersicht der Entwicklung der funktionalen Sprachen bzw. der Konzepte des funktionalen Programmierparadigmas geben:

Die Entwicklung funktionaler Sprachen wurde mit dem Lambda-Kalkül von Church [Chu33, Chu41, CR36] eingeleitet. Die erste funktionale Sprache -*Lisp*- ist von McCarthy [McC60] definiert worden, wobei die einzige Gemeinsamkeit mit Churchs Lambda-Kalkül in der Funktionsnotation bestand. Die später definierten funktionalen Sprachen waren enger mit dem Lambda-Kalkül verbunden. Heute steht der Lambda-Kalkül im Mittelpunkt der Konzeption der Funktionsdefinition und Anwendung jeder modernen funktionalen Sprache (z. B. *Haskell* und *Miranda*[2]). Die wichtigsten Charakteristika von McCarthys *Lisp* waren:

- Benutzung bedingter Ausdrücke und ihre Verwendung in Definitionen der rekursiven Funktionen
- Listen als Datenobjekte und Möglichkeit der Definition von Operationen höherer Ordnung über Listen
- *cons cell* als die grundlegende Struktur einer Liste und *garbage collection* als eine Methode für die Deallozierung des besetzten Speichers
- Listen repräsentieren sowohl Programme als auch Daten.

Eine rein-funktionale Version von Lisp ist im Jahr 1980 von Henderson [Hen80] definiert worden - Lispkit Lisp. Es gibt heute mehr als 20 inkompatible *Lisp*-Dialekte, von denen *Common Lisp*, *Eulisp* und *Scheme* die bedeutendsten sind. Die Sprache *Common Lisp* ist eine der funktionalen Sprachen, die ihre Anwendung in breiterem Maße auch in der Industrie gefunden hat. *Scheme* hat von *ISWIM* die Klarheit der Behandlung und von Church's Lambda-Kalkül die Kombinierbarkeit der Funktionen übernommen. Die wichtigsten Charakteristiken von *Scheme* sind:

- Behandlung der Funktionen als Objekte erster Ordnung
- *lexical scoping* der Identifikatoren.

[2] *Miranda* ist ein eingetragenes Warenzeichen von *Research Software Ltd.*

Die erste Definition von *Scheme* ist im Jahr 1975 von G. Sussman und G Steele Jr. gegeben worden. Die Definition dieser Sprache wurde bis heute mehrfach überarbeitet. Die neuste Version findet man in [RE91].

Im Jahr 1966 hat Landin die Sprache *ISWIM* (**I**f You **S**ee **W**hat **I** **M**een) definiert. Neue Konzepte, die diese Sprache dem funktionalen Paradigma brachte, sind: Infixe Syntax, *let-* und *where* -Bindungen, *off-side* basierte Identifizierung für Deklarationen und Ausdrücke, referentielle Transparenz.

Im Jahr 1962 ist die Sprache *APL* (**A** **P**rogramming **L**anguage) von Iverson [Ive62] definiert worden. Das ist keine rein-funktionale, sondern eine algebraische Sprache für *Arrays*. Später, im Jahr 1986, wurde eine *APL*-ähnliche Sprache definiert. Die Sprache heißt *FAC* [TP86] und ist rein-funktional, mit der Möglichkeit der Bearbeitung unendlicher *arrays*. Diese Sprache hat nicht-strikte Semantik.

Im Jahr 1978 hat Backus (einer der Entwickler der Sprache *Fortran*) die funktionale Sprache *FP* definiert. Es war die erste funktionale Sprache, die eine größere Aufmerksamkeit in den Computerwissenschaften erregt hatte. Die Programme in der Sprache *FP* sind sehr knapp gehalten und manchmal unverständlich. Die Sprache *FP* ist nicht an Churchs Lambda-Kalkül orientiert. Den mit dem Lambda-Kalkül verbundenen Stil hat später auch Backus bei der Definition seiner neuen Sprache *FL* akzeptiert.

Die Sprache *ML* wurde als ein Teil des von der Forschungsgruppe von M. Gordon definierten *proof-generating system* entwickelt [GMM+78]. *ML* wurde danach unter Einfluß der Sprache *Hope* [BMS80], die R. M. Burstall definiert hat, erweitert. Die letzte Version von *ML* heißt *Standard ML* bzw. *SML* [Mil84]. *ML* ist eine funktionale Sprache mit strikter Semantik [MTH89], mit der Möglichkeit der Definition der Ein- bzw. Ausgabe (*I/O System*), mit Programmoduln und sogenannten Referenzen (das sind Verweise auf den Speicher). Durch die Referenzen und das *I/O System* konnten Seiteneffekte verursacht werden, so daß *ML* keine rein-funktionale Sprache ist. Die durch *ML* eingeführten Möglichkeiten sind:

- Statische Typisierung, wobei der Polimorphismus im Sinne der Definition der Funktionen ohne Typabhängigkeit realisiert ist
- Definition neuer Datentypen
- *Pattern Matching*.

Die Sprache *ML* ist heutzutage weit verbreitet. Später erschien eine nicht-strikte Version von *ML* - *LML* (*Lazy ML*).

Im Ergebnis der Arbeit an einer erweiterten Ausdrucksmöglichkeit der funktionalen Programmierung von David Turner sind drei funktionale Sprachen entstanden, auf denen die moderne Schule der funktionalen Programmierung basiert: *SASL* [Tur76], *KRC* [Tur81] und *Miranda*[3] [Tur85].

[3]Das ist eine der kommerziellen funktionalen Sprachen (vielleicht die einzige).

In *SASL* (**S***t.* **A***ndrews* **S***tatic* **L***anguage*) geschriebene Programme haben eine mathematische Form, wobei die bedingten Ausdrücke durch Bedingungen neben den Ausdrücken stehen und dabei Auskunft geben, wann die Ausdrücke verwendbar sind. *SASL* hat eine nicht-strikte Semantik, und die Sprache ist durch Kombinatoren implementiert worden. Das war die erste auf der kombinatorischen Logik basierende Implementation einer funktionalen Sprache. Ein weiteres wichtiges Konzept von *SASL* war der *Currying*.

Die Sprache *KRC* ging aus einer Erweiterung der Sprache *SASL* hervor. Mit der Sprache wurden die Zermelo-Frenkel Ausdrücke (*ZF-expressions*) in das funktionale Paradigma eingeführt. Die Sprache *KRC* wiederum wurde zu *Miranda*^{TM} erweitert. *Miranda* ist stark typisiert (durch das Typsystem von *Hindley* und *Milner*) und erlaubt die Definition von Benutzer-definierten abstrakten und konkreten Typen.

Die funktionale Sprache, in welche alle Erfahrungen des Gebietes eingeflossen sind, ist *Haskell*. *Haskell* ist eine universelle, rein-funktionale, nicht-strikte Programmiersprache. Zu nennen sind folgende Innovationen, Konzepte und Charakteristika, über die *Haskell* verfügt:

- Funktionen höherer Ordnung
- *lazy evaluation*
- *static polymorphic typing*
- Benutzer-definierte Datentypen
- *pattern matching*
- *list-comprehensions*
- Moduln
- rein funktionales *I/O* System
- ein großes Angebot an vordefinierten Datentypen (Listen, *arrays*, *integers*, *reals*).

Haskell wurde 1990 definiert. Es gibt heute 3 bekannten Implementationen dieser Sprache, die an der Universitäten von Glasgow [oCS93], Yale [YU93] und Chalmers [Aug93] realisiert worden sind. Mehrere größen Programmierprojekte sind schon in *Haskell* implementiert worden [CGar, WR95, Sin90, Car93].

Zusammenfassend kann auf die große Vielfalt im Laufe der Zeit entwickelter funktionaler Sprachen hingewiesen werden, mit recht unterschiedlichem syntaktischem und semantischem Erscheinungsbild und immer ausgefeilteren speziellen Konzepten. Für die Bearbeitung unserer Thematik bedeutet das, eine Auswahl zu treffen und die Transformation funktionaler Sprachelemente schrittweise zu untersuchen.

1.3 Implementationstechniken für funktionale Sprachen

Gegenstand der vorliegenden Dissertation ist die Transformation funktionaler Sprachen in prozedurale. Ein derartiges Vorgehen ist damit Teil einer allgemeinen Implementationsstrategie für funktionale Sprachen.

Hier sind Querbeziehungen zu herkömmlichen Implementationstechniken für funktionale Sprachen zu ziehen, Vergleiche vorzunehmen und Möglichkeiten für die Kombination der Ansätze zu untersuchen.

In diesem Abschnitt geben wir einen kurzen Überblick über bekannte Implementationstechniken für funktionale Sprachen, während wir im Kapitel 2 auf diesen Gegenstand näher eingehen.

Die grundlegenden Techniken der Implementation funktionaler Programmiersprachen basieren auf folgenden Strategien:

- Interpreter-basierte Implementationen:
  Die funktionale Sprache wird in einen Zwischencode übersetzt, der danach bis zur unreduzierbaren Form (*weak head-normal form*) interpretiert (reduziert) wird.

- Stack-basierte maschinelle Implementationen:
  Übersetzung des Quellprogramms in die Sprache einer abstrakten Maschine, deren Zustand mit Hilfe von Registern dargestellt wird. Die Evaluation des Maschinencodes wird durch Registermanipulation ausgeführt.

- Graph-Reduktion-basierte Implementationen:
  Bindungen der formalen an die aktuellen Parameter werden mit Zeigern realisiert. Reduzierbare Ausdrücke werden mit ihren Äquivalenten in „nicht mehr reduzierbarer Form" überschrieben.

- Implementation durch Quelltexttransformation:
  Die durch Übersetzung in eine andere höhere Programmiersprache realisierte Implementation.

Die Interpreter-basierten Implementationstechniken ziehen wir wegen ihrer Ineffizienz nicht in Betracht. Wir werden die Graph-Reduktion-basierten (SK-Kombinator-Reduktion und die abstrakten *G*- und *STG*-Maschine), die Stack-Manipulation-basierten Implementationen und die direkten Übersetzungen analysieren.

Im Kapitel 2 wird jede Technik näher beschrieben.

1.4 Quelltexttransformation als Implementationstechnik

„Source to Source" Transformation bzw. Quelltexttransformation bedeutet eine Transformation, deren Ein- und Ausgabe Quelltexte verschiedener höherer Programmiersprachen sind. Dieses Implementationsverfahren ist mittlerweile weitverbreitet, wobei zunächst nur prozedurale Sprachen einbezogen worden waren [BHHW89, Cro88, Owe87, Per88, Pla89, Pri92]. In jüngster Zeit werden aber auch funktionale Programmiersprachen auf diese Weise implementiert. Die Vorteile dieser Technik bestehen in der Portabilität und Kompatibilität von Software, was heutzutage angesichts steigender Softwarekosten von besonderer Bedeutung für Softwareentwicklung geworden ist.

In diesem Abschnitt gehen wir zunächst auf die Vorteile der Implementation höherer Programmiersprachen durch Quelltexttransformation ein. Daran anschließend werden Projekte zur Quelltexttransformation zusammengefaßt, die funktionale Programmiersprachen betreffen und die in diesen Projekten auftretenden Probleme benannt.

Abschließend gehen wir auf die Ziele der vorliegenden Dissertation und den Aufbau der Arbeit ein.

1.4.1 Vorteile der Implementation durch Übersetzung in eine andere höhere Programmiersprache

Quelltexttransformation stellt ein Compilationsschema dar, dessen Ein- und Ausgabe Programme zweier höherer Sprachen sind. Daraus ergeben sich folgende Vorteile:

- Portabilität der Software mit der Möglichkeit, die entwickelten Programme auf verschiedene Rechnerarchitekturen zu überführen
- Kompatibilität der Software im Sinne einer Verbindungsmöglichkeit mit Programmen anderer Sprachen
- Weiterentwicklungsmöglichkeiten des erzeugten Zielcodes
- Neue Debuggingsmöglichkeiten:
  - Debugging des Zielprogramms mit Hilfe der dort vorhandenen Debuggingwerkzeuge
  - Einbau von speziellen Debuggingroutinen in das Zielprogramm
- Einfacheres Compilationsschema:
  Eine Compilation in eine höhere Programmiersprache ist leichter zu realisieren als eine Compilation bis auf das Niveau einer Maschinensprache.

- Verbindung unterschiedlicher Programmierparadigmen:
  Heutzutage wird versucht, das funktionale und das prozedurale Paradigma näher zu bringen. In [ACD+97, CSS95] werden funktionale Konzepte in prozedurale Sprachen eingebaut. Es werden auch funktionale Sprachen mit deklarativen Elementen (z. B. *Scheme* und *ML*) definiert. In [Feh89] werden die zwei unterschiedlichen Paradigmen auf folgende Art und Weise verbunden: Es wird eine denotationale Semantik einiger Konstrukte imperativer Sprachen definiert und in Verbindung mit der Semantik funktionaler Sprachen gebracht.

  Die Transformation einer funktionalen Sprache in eine prozedurale Sprache kann als eine Brücke zwischen den beiden Paradigmen angesehen werden.

1.4.2 Ansätze zur Quelltexttransformation funktionaler Programmiersprachen

Funktionale Programmiersprachen wurden in einer Reihe moderner Compilerprojekte durch Quelltexttransformation implementiert.

Zu nennen sind folgende Implementationen:

- Haskell $\Leftrightarrow\!\!\rightarrow$ C [oCS93]:
  Die Sprache *Haskell* wird in die Sprache *STG* übersetzt, die die Sprache der abstrakten *STG*-Maschine ist. Die *STG*-Sprache wird danach in die Sprache *C* übersetzt.

- Scheme $\Leftrightarrow\!\!\rightarrow$ C [Bar93], [Tam93]:
  Hierbei handelt es sich um zwei Projekte, bei denen die Sprache *Scheme* direkt in die Sprache *C* ohne abstrakte Maschine übersetzt wird. Der in [Tam93] definierte Compiler unterstützt nicht alle Sprachkonzepte von *Scheme*. Im erzeugten Zielcode haben alle Objekte den gleichen Typ.
  Der von Joel Bartlett implementierte Compiler [Bar93] ist ein effizienter Compiler für *Scheme*. *Scheme*-Funktionen werden direkt in C-Funktionen übersetzt, wenn es möglich ist. Ansonsten wird für eine Funktion ein *Closure* erzeugt (z. B. bei Funktionen höherer Ordnung). Auch hier sind nicht alle Sprachkonzepte von *Scheme* realisiert.

- Haskell $\Leftrightarrow\!\!\rightarrow$ Common Lisp [YU93]:
  Die Sprache *Haskell* wird in eine Zwischensprache (eine einfachere Form von *Haskell*) übersetzt, woraus der Zielcode in *Lisp* erzeugt wird. Die Implementation basiert nicht auf einer abstrakten Maschine.

- Haskell $\Leftrightarrow\!\!\rightarrow$ LML [Aug84, Aug93]:
  Die Sprache *Haskell* wird in die Sprache *LML* übersetzt, die ebenfalls eine

nicht-strikte Sprache ist. Die Implementation von *LML* basiert auf der Übersetzung dieser Sprache in die Sprache der abstrakten *G*-Maschine [Aug84]. Bei dieser Implementation von *Haskell* spielt die Sprache *LML* die Rolle einer nicht-strikten Zwischensprache. Der Code der abstrakten *G*-Maschine wird danach in eine Assemblersprache übersetzt.

- SML $\Leftrightarrow\!\!\rightarrow$ C [TAL90]:
  Es handelt sich hierbei um eine auf *Continuations* basierende Übersetzung von *SML* nach *C*, wobei die Sprache *C* als eine Assemblersprache verwendet wird. Die Implementation basiert auf dem in [App92] angegebenen Verfahren der Übersetzung von *SML* in eine Assembler-ähnliche Sprache (*CPS*).

- Opal $\Leftrightarrow\!\!\rightarrow$ C [DFG$^+$94]:
  Für die an der Technischen Universität Berlin entwickelte algebraische Sprache Opal wurde eine effiziente Implementation vorgenommen. Die Implementation basiert auf einer siebenstufigen Übersetzung der Quellsprache in eine imperative Form. Der letzte Übersetzungsschritt erzeugt den Zielcode in der Sprache C.

- Implementation des Vektor-basierten Parallelismus in C und Fortran 90 [ACD$^+$97, CSS95, Jä96]:
  Die Sprachen C und Fortran 90 wurden mit einigen funktionalen Elementen erweitert, um den V-geschachtelten Parallelismus ausdrücken zu können. Die neu definierten funktionalen Elemente werden in ein prozedurales Modell überführt.

- CIP-Projekt [BEH$^+$87, BBM$^+$85]:
  Es wurde ein System von Transformationen einer funktionalen Programmiersprache bis hin zu einer prozeduralen Form definiert. Jeder Transformationsschritt wird durch formale Transformationsregeln beschrieben, wodurch die Korrektheit bewiesen werden kann.

- Transformationen innerhalb funktionaler Sprachen [Rie91]:
  In diesem Projekt am MIT ging es um Transformationen zwischen unterschiedlichen funktionalen Sprachen mit dem Ziel, die Leistungsfähigkeit ihrer Sprachkonzepte miteinander zu vergleichen.

Darüber hinaus gibt es noch weitere Quelltextcompiler funktionaler Sprachen: Common Lisp $\Leftrightarrow\!\!\rightarrow$ C [HK93, GBH$^+$96], EuLisp $\Leftrightarrow\!\!\rightarrow$ C [eul94], Sather $\Leftrightarrow\!\!\rightarrow$ C [ICSIiB92], Miranda^{TM} $\Leftrightarrow\!\!\rightarrow$ Haskell [How92].

Alle durch Quelltexttransformation realisierten Implementationen können in zwei Kategorien unterteilt werden:

1. Implementation durch direkte Übersetzung der funktionalen Sprache in eine andere Sprache z. B. [Bar93, How92, HK93, GBH+96, eul94]

2. Auf einer abstrakten Maschine basierende Implementation, die im allgemeinen in zwei Schritten umgesetzt wird: Zunächst erfolgt eine Übersetzung in den Code einer abstrakten Maschine, der anschließend in eine prozedurale Sprache überführt wird [PJ92, Aug93, oCS91]. Da die Unterschiede zwischen funktionalen Sprachen und konventionellen Maschinensprachen erheblich sind, werden spezielle abstrakte Maschinen definiert, um alle Sprachkonzepte einer funktionalen Sprache implementieren zu können.

Unsere weitere Analyse der Quelltextcompiler basiert auf dieser Unterteilung. Darüber hinaus beschränken wir uns auf Quelltextcompiler, die funktionale in prozedurale Sprachen übersetzen.

1.4.3 Probleme der Quelltexttransformation für funktionale Sprachen

Trotz der in Abschnitt 1.4.1 genannten Vorteile der Implementation höherer Programmiersprachen durch Quelltexttransformation weisen die auf dieser Technik vorgenommenen Implementationen für funktionale Sprachen eine Reihe von Problemen auf. Das betrifft sowohl die direkte Übersetzung als die Verwendung einer abstrakten Zwischensprache.

Die direkte Übersetzung einer funktionalen Sprache in eine prozedurale Sprache ist nicht einfach. Insbesondere stellt sich die Frage, in welchem Maße die Vielzahl höherer Sprachkonstruktionen funktionaler Sprachen zu erfassen ist. So bereitet beispielsweise die Übersetzung folgender funktionaler Konzepte besondere Probleme: Funktionen höherer Ordnung, *curried functions*, Funktionen als Objekte erster Ordnung ... usw.

Auf direkter Übersetzung basierende Quelltextcompiler sollten sich durch Transparenz der Transformation und Lesbarkeit des Zielcodes auszeichnen. Auf diese Weise könnte auch das Zielprogramm in ein Debugging des Quellprogramms einbezogen, weiterentwickelt und modifiziert werden. Als Nachteile bisher realisierter auf direkter Übersetzung basierender Quelltextcompiler sind folgende zu nennen:

- Der erzeugte Zielcode ist schlecht lesbar. Damit werden die Möglichkeiten der Weiterentwicklung und Modifikation des Zielprogramms eingeschränkt.

- Die Beziehung zum Originalprogramm ist nicht transparent genug
  Das verursacht folgende negative Effekte:

  - Ein Debugging auf der Ebene des Zielcodes wird erschwert

  - Die Sprachkonstruktionen der Zielsprache werden nur in eingeschränktem Maße verwendet.

Bei Verwendung abstrakter Maschinen ergibt sich ein weiteres Problem:

- Die zugrunde liegenden abstrakten Maschinen sind in erster Linie für die Implementation der für die funktionalen Sprachen spezifischen Konzepte definiert worden. Ein mögliches ,,back end" der Maschine spielt dabei keine Rolle, d. h. Aspekte der Erzeugung des Codes in einer höheren Programmiersprache werden nicht untersucht.

1.5 Zielstellung und Ergebnisse der Arbeit

Zielstellung

Die vorliegende Arbeit soll einen Beitrag zur Problematik der Entwicklung von Compilern funktionaler Sprachen zur Erzeugung von Programmen in prozeduralen Sprachen liefern. Analysiert werden sowohl Techniken der direkten Übersetzung in eine prozedurale Sprache als auch Techniken, die auf abstrakten Maschinen basieren.

Ziele der Arbeit, die sich auf die Techniken der direkten Übersetzung beziehen, sind:

- Definition einer Transformation, die eine funktionale Sprache in eine prozedurale Sprache direkt und transparent überführt und Implementation eines der definierten Transformation entsprechenden Compilers. Der Compiler wird mit dem in [Bar93] vorgestellten Compiler verglichen. Analysiert werden folgende Aspekte:

  - Möglichkeiten der weiteren Bearbeitung der mit den Compilern erzeugten Programme
  - transparente Beziehung zwischen Originalprogramm und erzeugtem Zielprogramm
  - Möglichkeiten der Verbindung mit anderen Programmen
  - Effizienz des Zielcodes.

  Dieser Compiler kann auch als ein Compiler für die Entwicklung prozeduraler Programme betrachtet werden.

Bezüglich der Verwendung abstrakter Maschinen werden folgende Ziele definiert:

- Untersuchung der Implementationsmöglichkeit eines auf einer bestimmten konventionellen Technik basierenden Compilers mit einem ,,back-end" für die Erzeugung prozeduralen Codes. Dabei werden notwendige Modifikationen der Implementationstechnik einbezogen. Zwei Techniken werden näher betrachtet:

– Stack-basierte Implementationstechnik (*SECD*-abstrakte Maschine)

– Implementation durch die SK-Graph-Reduktion

Ergebnisse der Arbeit

Die Ergebnisse der Arbeit liegen im Bereich der Entwicklung von Compilern funktionaler Sprachen zur Erzeugung von Zielcode in einer prozeduralen Programmiersprache. In der vorliegenden Dissertation werden mehrere Implementationstechniken funktionaler Sprachen analysiert, wobei für jede Technik ein Verfahren zur Realisierung eines auf der Übersetzung in eine prozedurale Sprache basierenden Compilers angegeben wird. Durch die verschiedenen Implementationsverfahren, die in der Arbeit analysiert bzw. definiert wurden, sind die beiden großen Klassen funktionaler Sprachen abgedeckt worden:

- strikte funktionale Programmiersprachen

- nicht-strikte funktionale Programmiersprachen

Darüber hinaus wird in der Arbeit ein Compiler für eine einfache strikte funktionale Programmiersprache (erweiterter Lambda-Kalkül), der Zielcode in einer prozeduralen Zielsprache direkt erzeugt, definiert und seine Implementation näher beschrieben. Dabei ist die Übersetzung weitestgehend transparent.

Im folgenden wird ein zusammenfassender Überblick über die Ergebnisse der Arbeit angegeben.

Die Dissertation kann in drei größere Teile gegliedert werden, die sich mit folgenden Bereichen beschäftigen:

1. Direkte Übersetzungen funktionaler in prozedurale Programmiersprachen

2. Übersetzung des Codes der abstrakten *SECD*-Maschine in eine prozedurale Sprache

3. Erzeugung des Zielcodes in einer prozeduralen Programmiersprache bei Graph-Reduktion-basierten Implementationstechniken.

Im ersten und zweiten Teil wird die Implementation der strikten Sprachen analysiert. Der dritte Teil bezieht sich auf die Implementation der nicht-strikten funktionalen Sprachen.

Als grundlegender Beitrag der vorliegenden Dissertation können somit

- die Zusammenfassung, Systematisierung und der Vergleich existierender Ansätze zur Quelltexttransformation funktionaler Programmiersprachen [oCS93, Bar93, Tam93, YU93, Aug84, Aug93, TAL90, DFG+94, ACD+97, CSS95, Jä96]

- die Definition und Implementation eines Compilers, der auf einer direkten Übersetzung einer funktionalen in eine prozedurale Sprache basiert und
- die Verbindung der Quelltexttransformation und traditioneller Implementationstechniken für funktionale Sprachen (Stack-basierte Implementationen [Lan64], Graph-Reduktion-basierte Implementationen [Wad71, Dil88])

angesehen werden.

Direkte Übersetzung funktionaler in prozedurale Programmiersprachen

In diesem Teil der Dissertation werden zunächst konzeptionelle Unterschiede zwischen funktionalen und prozeduralen Programmiersprachen analysiert und ein Algorithmus für die Übersetzung eines erweiterten Lambda-Kalküls in eine prozedurale Sprache angegeben. Die Transformation realisieren wir in einer 4-stufigen Übersetzung:

- *Lifting* aller Lambda-Abstraktionen auf die umgebende Ebene des Programms
- Transformationen innerhalb der funktionalen Quellsprache
- Übersetzung der funktionalen Quellsprache in eine prozedurale Zwischensprache
- Übersetzung der funktionalen Zwischensprache in eine konkrete prozedurale Sprache.

Mit dem ersten Schritt werden alle Lambda-Abstraktionen auf der umgebenden Programmebene als Funktionen definiert und ihnen ein Name zugeordnet. Im Programm wird auf die Namen der Funktionen verwiesen.

Durch Transformationen innerhalb der funktionalen Quellsprache wird eine Form des funktionalen Programms angestrebt, von der aus die nachfolgende Übersetzung erleichtert wird. Die Überführung der funktionalen Quellsprache in die prozedurale Zwischensprache wird im dritten Schritt der Übersetzung durchgeführt.

Die weitere Übersetzung der prozeduralen Zwischensprache in eine konkrete prozedurale Programmiersprache ist einfach. Die Programme der Zielsprache unterstützen einfache Typen der Sprache. In diesem Teil ist auch ein der definierten Übersetzungsfunktion entsprechender Algorithmus für die Zuordnung der Typen an die Variablen und Funktionen der Zielsprache definiert. Der Algorithmus basiert auf dem Typprüfungalgorithmus von Milner [Mil78]. Als Beispiel wird die Übersetzung nach *Modula-2* angegeben.

In diesem Teil der Arbeit sind folgende Ergebnisse im Vergleich zu ähnlichen Ansätzen aus der Literatur hervorzuheben:

- Ein Großteil der Ansätze aus der Literatur zur Übersetzung funktionaler in prozedurale Sprachen basiert auf dem *Continuation-Passing-Style* [App92], [KKR+86], [TAL90]. Bei dieser Technik entstehen prozedurale Programme, die nicht dem prozeduralen Paradigma auf natürliche Weise folgen.

  Unsere Implementation ist keine *Continuation-Passing-Style*-basierte Implementation. Funktionen der funktionalen Quellsprache werden in Funktionen der prozeduralen Zielsprache übersetzt. Das ist der wesentliche Unterschied des von uns entwickelten Compilers vom Scheme-C-Compiler von Kranz [KKR+86] und vom SML-C-Compiler von Tarditi, Acharya und Lee [TAL90], die auf *Continuations* basieren.

- Unser Transformationsschema kann den Grundprinzipien nach mit dem Scheme-C-Compiler [Bar93], der ebenfalls nicht die Technik des *Continuation-Passing-Style* zugrunde legt, verglichen werden.

  Bei diesem Compiler wird das funktionale Quellprogramm in Codesequenzen unterteilt, die mit go-to-Anweisungen verbunden werden. Bei unserem Compiler wird der funktionale Quellcode in die Zielsprache transparent übersetzt und damit der Programmablauf adäquat überführt.

- In den von Bartletts Compiler erzeugten Programmen haben alle Variablen den gleichen Typ, der inhaltlich eine strukturierte Vereinigung aller möglichen Typen sein muß. Unser Compiler unterstützt einfache Typen der prozeduralen Zielsprache, was die Lesbarkeit und Effizienz der erzeugten Zielprogramme verbessert.

- Vergleiche unseres Compilers mit dem von Bartlett ergeben Laufzeitgewinne zwischen 4 und 70% für die mit unserem Compiler übersetzten Programme. Diese Laufzeitgewinne gründen sich auf die natürliche Transformationsstrategie.

- Im Gegensatz zu anderen Implementationen [Bar93, KKR+86, TAL90] liegt unserem Compiler ein formales System von Transformationsregeln zugrunde, in das wir z. T. bekannte theoretische Ansätze unterschiedlicher Arbeiten einbezogen haben (Lambda-Lifting [Aug84, Jon87], Transformationen von Santos [San95]), z. T. darüber hinausgehende spezielle Regeln selbst entwickelt haben.

Übersetzung des Codes der abstrakten *SECD*-Maschine in eine prozedurale Sprache

In diesem Teil der Dissertation wird die Problematik einer optimalen Realisierung der operationellen Semantik der abstrakten *SECD*-Maschine in einer prozeduralen Programmiersprache analysiert. Zunächst wird das Laufzeitsystem eines *SECD*-

Evaluationssystems definiert. Für die Erzeugung des Zielprogramms in einer prozeduralen Programmiersprache geben wir zwei Möglichkeiten an:

- Simulation von *Transitions* der *SECD*-Maschine in der prozeduralen Zielsprache
- Erzeugung des prozeduralen Programms für die Manipulation der *S*-, *E*- und *D*-Register (ohne *C*-Register).

Die erste Möglichkeit ist eine immer anwendbare Lösung für Compiler dieser Art: In jedem Evaluationsschritt wird ein Maschinenbefehl vom *C*-Register genommen, und eine ihm entsprechende Manipulation mit den Maschinenregistern wird vorgenommen. Die zweite Lösung besteht aus der Erzeugung einer sequentiellen Folge von Maschinenbefehlen in der prozeduralen Zielsprache. Diese Implementation ist effizienter, und das Laufzeitsystem besteht aus den Registern *S*, *E* und *D*. Die Übersetzungsfunktion, die den *SECD*-Code in den Code einer prozeduralen Sprache übersetzt und die für sie notwendigen Hilfsfunktionen werden auch in diesem Teil der Dissertation definiert.

Das in diesem Kapitel beschriebene Implementationsverfahren ist eine konkrete Realisierung der abstrakten *SECD*-Maschine von Landin [Lan64] (call-by-value). Mit dieser Implementation haben wir gezeigt, wie eine strikte funktionale Sprache in eine prozedurale Sprache übersetzt werden kann, so daß die Funktionen höherer Ordnung, Lambda-Abstraktionen und curried-Funktionsapplikation korrekt und ohne größere Probleme implementiert werden können. Diese Implementationstechnik ist grundsätzlich langsamer als die Übersetung mit *Continuations* [KKR+86, TAL90]. Der Vorteil unseres Verfahrens besteht jedoch in einer einfacheren Implementation und der Möglichkeit der Modifizierung der Implementation auch für nicht-strikte Programmiersprachen.

Erzeugung des Zielcodes in einer prozeduralen Programmiersprache bei Graph-Reduktion-basierten Implementationstechniken

In diesem Teil wird die Erzeugung des Zielcodes in einer prozeduralen Programmiersprache bei den Graph-Reduktion-basierten Implementationstechniken funktionaler Programmiersprachen behandelt.

Als Beispiel des Graph-Reduktors wurde die abstrakte SK-Maschine implementiert. Das Verfahren ist bei den abstrakten *G*- und *STG*-Maschinen auch anwendbar.

Nach der Definition des Laufzeitsystems und des Graph-Evaluators wird diese Implementationstechnik mit dem im ersten Teil der Dissertation definierten Transformationssystem zusammengesetzt: Strikte Funktionen der funktionalen Quellsprache werden *direkt* in die Zielsprache übersetzt. Mit der neuen Menge der Kombinatoren und zu dem Laufzeitsystem gehörenden Funktionen werden direkte Aufrufe der in der Zielsprache erzeugten Funktionen realisiert.

Neben den neuen Kombinatoren sind auch neue Übersetzungsfunktionen, die die Kommunikation des Graph-Evaluators und der direkt erzeugten Funktionen in der Zielsprache unterstützen, definiert.

Unser entwickelter Compiler kann als eine effizientere Realisierung des in [Dil88] definierten Lispkit-Lisp-Interpreters betrachtet werden. Im Vergleich mit dem in [LL92] definierten Verfahren der Implementation einer nicht-strikten funktionalen Programmiersprache mit einer Erzeugung prozeduraler Zielprogramme ist unser Verfahren einfacher zu implementieren und zu verstehen. Die in [LL92] modifizierte Implementationstechnik mit kategorischen Kombinatoren ist effizienter als die SK-Kombinatoren-basierte Implementationstechnik. In [LL92] ist nicht geklärt, wie die strikten Funktionen der funktionalen Quellsprache in Funktionen der prozeduralen Zielsprache übersetzt werden.

1.6 Aufbau der Arbeit

Wir beginnen im Kapitel 2 mit einer Übersicht zu den Implementationstechniken funktionaler Programmiersprachen. Zwei Implementationstechniken werden ausführlicher beschrieben (abstrakte *SECD*-Maschine und *SK*-Graph-Reduktion) und zwei weitere nur im Überblick (Interpreter-basierte Implementationstechnik und abstrakte *G*-Maschine) erläutert.

Im Kapitel 3 werden zunächst Probleme einer direkten Übersetzung einer funktionalen in eine prozedurale Sprache diskutiert. Daran anschließend wird ein konkreter Übersetzungsalgorithmus angegeben, für den alle Transformationsschritte durch Transformationsregeln eindeutig definiert werden. Die auf dieser Grundlage von uns vorgenommene Implementation wird in den Grundzügen beschrieben und abschließend mit einem verwandten Compiler inhaltlich verglichen.

Kapitel 4 bezieht sich auf die Implementation einer funktionalen Sprache durch die abstrakte *SECD*-Maschine. Es wird ein Algorithmus für die Übersetzung des *SECD*-Codes in eine prozedurale Sprache definiert.

Eine auf der SK-Graph-Reduktion basierende Implementation wird im Kapitel 5 betrachtet. Es wird ein Verfahren zur Erzeugung von Zielcode mit Aufrufen des Maschinenevaluators (bei derartigen Implementationen die einzige Lösung) angegeben. Aufgrund der Besonderheit des betrachteten SK-Reduktionsystems (Erzeugung des Zielcodes in einer höheren prozeduralen Programmiersprache) wird ein Verfahren für eine Optimierung des Compilers betrachtet.

Kapitel 6 gibt eine Zusammenfassung der Arbeit und zeigt Möglichkeiten weiterer Forschungsaktivitäten auf.

Kapitel 2

Implementationstechniken funktionaler Programmiersprachen

Im Rahmen der vorliegenden Dissertation wird die Implementation funktionaler Programmiersprachen durch Erzeugung des Zielcodes in einer prozeduralen Programmiersprache untersucht. Da eine direkte, transparente Übersetzung wegen der großen Unterschiede zwischen beiden Programmierparadigmen (prozedurales und funktionales) schwer durchführbar ist, sollen auch die konventionellen Implementationstechniken funktionaler Programmiersprachen einbezogen werden. Diese Techniken werden daraufhin untersucht, ob Möglichkeiten zur Erzeugung von Code in einer höheren prozeduralen Programmiersprache existieren.

In diesem Kapitel wird eine Übersicht über Implementationstechniken funktionaler Sprachen gegeben.

Einige Techniken werden eingehender analysiert (Stack-basierte Implementationen, SK-Graph-Reduktion), andere nur am Rande erläutert (Interpretersysteme, abstrakte G-Maschine). Weitere Techniken, die entweder nicht so große Bedeutung in der Implementationstheorie funktionaler Sprachen gehabt hatten oder unsere Arbeit nicht weiter beeinflußt haben, werden nur genannt und mit den anderen Techniken verglichen.

Die Implementationstechniken funktionaler Sprachen können folgendermaßen unterteilt werden:

- Interpreter-basierte Implementationen [McC60]:
  Die funktionale Sprache wird in einen Zwischencode übersetzt, der danach bis zur unreduzierbaren Form (*weak head-normal form*, Bezeichnung: *WHNF*) vereinfacht wird. Dieser Zwischencode hat meistens die Form eines erweiterten Lambda-Kalküls.

- Stack-basierte maschinelle Implementationen [Hen80]:
  Das Quellprogramm wird in eine Sprache einer abstrakten Maschine übersetzt,

deren Zustand mit Hilfe von Registern dargestellt wird. Die Evaluation des Maschinencodes wird durch Registermanipulation umgesetzt. In dieser Arbeit wird die abstrakte *SECD*-Maschine als Beispiel dieser Technik analysiert.

- Graph-Reduktion-basierte Implementationen [Jon87]:
  Die Bindung der formalen an die aktuellen Parameter wird mit Pointern realisiert. Reduzierbare Ausdrücke werden mit ihren Äquivalenten in *WHNF* ersetzt. Die Repräsentation des gesamten funktionalen Programms hat die Form eines Graphen. Am Anfang wurden die funktionalen Sprachen in einen Lambda-Ausdruck übersetzt und die Operationen über ihm realisiert. Danach wurde diese Lösung mit der kombinatorischen Logik verbunden: Kombinatorische Darstellung des funktionalen Programms und seine Evaluation.

  Die funktionale Sprache wird in eine Form übersetzt, die keine freien Variablen enthält, sogenannte kombinatorische Terme. Diese Terme werden nach der für jeden Kombinator definierten Reduktionsregel reduziert. Als Beispiel dieser Technik analysieren wir die SK-Graph-Reduktion.

- Fortgeschrittene Graph-Reduktion-basierte Techniken [Joh84, Aug84, PJ92]:
  Das sind Techniken, die eine Mischung der zwei letztgenannten Techniken darstellen. Die Programmevaluation wird durch Registermanipulationen durchgeführt. Die grundlegende Operation der Evaluation ist die Graph-Reduktion. Der Code der Maschine produziert den Graphen, der später reduziert wird. Das ist die effizienteste bekannte Implementationstechnik. In der vorliegenden Arbeit werden nur die Grundzüge der Technik beschrieben. Die bei den o. g. abstrakten SK- und SECD-Maschinen untersuchten Aspekte

  – Stack-Manipulation und
  – Graph-Reduktion

  werden in der vorliegenden Dissertation mit der Quelltexttransformation zwischen funktionalen und prozeduralen Sprachen in Verbindung gebracht. Da die abstrakte *G*-Maschine eine Kombination der beiden behandelten abstrakten Maschinen ist, wird damit die Möglichkeit der Erzeugung des prozeduralen Zielcodes bei den auf der *G*-Maschine basierenden Implementationen abgedeckt. Der Algorithmus der Übersetzung bzw. der Umsetzung der Konzepte der Stack-Manipulation in einem strikten prozeduralen Programmparadigma wird im Kapitel 4 definiert. Die Möglichkeit der Erzeugung des prozeduralen Zielcodes bei der auf der Graph-Reduktion basierenden Technik wird am Beispiel der SK-Graph-Reduktion basierenden abstrakten Maschine im Kapitel 5 behandelt.

- Die durch Übersetzung in eine andere höhere Programmiersprache realisierte Implementation (Quelltexttransformation):

Hierunter fassen wir alle Implementationen zusammen, die als Zielcode Code einer höheren Programmiersprache erzeugen. So können wir beispielsweise sagen: ,,durch eine Stack-basierte abstrakte Maschine realisierte Quelltexttransformation". Das bedeutet eine Implementation, die auf einer bestimmten abstrakten Maschine basiert und deren Zielcode in einer der höheren Programmiersprachen erzeugt wird.

- Andere Implementationstechniken:
  - Die auf Datenflußmaschinen basierenden Implementationen [GH86, EW88]. Diese Technik kann als ,,fleißige" Graph-Reduktions-Technik beschrieben werden.
  - *PAM*-Maschine (*Ponder Abstract Machine*) [FW86]: Nach Konstruktion (und Effizienz) sehr ähnlich der *G*-Maschine (fortgeschrittene Graph-Reduktion).
  - Auf Superkombinatoren basierende abstrakte *TIM*-Maschine [FW87]. [1]

Die auf Registermanipulation und Graph-Reduktion basierenden Techniken werden nachfolgend ausführlicher beschrieben.

In den Kapiteln 4 und 5 werden zwei der hier angegebenen Techniken auf ihre Möglichkeit zur Erzeugung prozeduralen Zielcodes untersucht.

Da der Lambda-Kalkül eine grundlegende Rolle bei der Implementation funktionaler Programmiersprachen spielt, erläutern wir zunächst die wesentlichen Begriffe dieses Formalismus, seine Syntax und operationelle Semantik.

2.1 Lambda Kalkül

Der Lambda-Kalkül ist ursprünglich unabhängig von den funktionalen Sprachen entwickelt worden [Chu33, Chu41, CR36]. Bei der Anwendung moderner funktionaler Sprachen ist das Prinzip des Lambda-Kalküls, also das Prinzip der ,,manipulierbaren Funktionen" ihre grundlegende Eigenschaft. Der Lambda-Kalkül verleiht den funktionalen Sprachen eine besondere Ausdruckskraft. Oftmals ist der erste Schritt der Implementation einer funktionalen Sprache die Übersetzung in eine Zwischensprache, die im wesentlichen ein *syntactic sugared* Lambda-Kalkül ist. Beispielsweise ist bei der SK-Graph-Reduktion-basierten Implementation der gesamte Prozeß der Evaluation eines funktionalen Programms eine Manipulation von Lambda-Ausdrücken.

Der Lambda-Kalkül stellt also häufig bei der Implementation höherer funktionaler Programmiersprachen ein Zwischenniveau dar und ist darüber hinaus für das Verstehen des Modells der funktionalen Programmierung von großem Interesse. Der

[1] Die Begriffe: Superkombinator, *lazy* (träge), *eager* (fleißig) werden später erläutert.

Lambda-Kalkül wird formal durch die Syntax des Lambda-Ausdrucks beschrieben und besitzt eine operationelle Semantik.

2.1.1 Syntax des Lambda-Ausdrucks

Lambda-Ausdrücke stellen die grundlegenden syntaktischen Konstruktionen des Lambda-Kalküls dar. In der Abbildung 2.1 ist die Syntax eines Lambda-Ausdrucks angegeben.

Abbildung 2.1: Grammatik eines Lambda-Ausdrucks

<*exp*>::=<*constant*>	Konstanten
| <*variable*>	Variablen
| (<*exp*> <*exp*>)	Funktionsapplikationen
| λ <variable>.<*exp*>	Lambda-Abstraktion

Die angegebene Grammatik des Lambda-Ausdrucks ist recht einfach. Für die Implementation funktionaler Sprachen wird dieser Kalkül um die Standardfunktionen der zu implementierenden Sprache sowie um *let*- und die rekursiven *let*- bzw. *letrec*-Konstruktionen erweitert. Lambda-Abstraktion bedeutet Definition einer neuen Funktion. Funktionsaufrufe haben eine präfixe Notation.

2.1.2 Operationelle Semantik des Lambda-Kalküls

Wir haben die Grammatik des Lambda-Kalküls angegeben, damit wir syntaktisch korrekte Ausdrücke des Lambda-Kalküls erzeugen können. Wie der Lambda-Kalkül „funktioniert" bzw. wie aus einem Lambda-Ausdruck ein anderer Ausdruck abgeleitet werden kann, zeigt uns seine operationelle Semantik. In diesem Abschnitt werden wir die Grundzüge der operationellen Semantik zusammenstellen, ohne in jedem Fall formalisierte Definitionen anzugeben.

Definieren wir zunächst, wann eine Variable x in einem Lambda-Ausdruck frei bzw. gebunden vorkommt.

Definition. Eine Variable kommt in einem Lambda-Ausdruck frei vor:

(i) x kommt frei in x vor.

(ii) x kommt frei in *(E F)* vor $\Leftrightarrow$ x kommt frei in E vor *oder* x kommt frei in F vor.

(iii) x kommt frei in $\lambda y.E$ vor $\Leftrightarrow$ x und y sind verschiedene Variablen *und* x kommt frei in E vor.

(iv) Eine Variable kommt nur frei in einem Lambda-Ausdruck vor, wenn das aus obigen Regeln **(i)**, **(ii)** und **(iii)** folgt.

Definition. Eine Variable kommt in einem Lambda-Ausdruck gebunden vor:

(i) x kommt gebunden in *(E F)* vor $\Leftrightarrow$ x kommt gebunden in E vor *oder* x kommt gebunden in F vor.

(ii) x kommt gebunden in $\lambda y.E$ vor $\Leftrightarrow$ (x und y sind dieselben Variablen *und* x kommt frei in E vor) *oder* x kommt gebunden in E vor.

(iii) Eine Variable kommt nur gebunden in einem Lambda-Ausdruck vor, wenn das aus obigen Regeln **(i)**, **(ii)** folgt.

Das Zeichnen „$\Leftrightarrow$" bedeutet *wenn und nur wenn.*

Beta-Konversion

Beta-Reduktion bedeutet die Auswertung einer Funktionsapplikation. Die Anwendung einer Lambda-Abstraktion auf aktuelle Argumente ist eine Instanz des Rumpfes der Lambda-Abstraktion, wobei alle freien Vorkommem des formalen Parameters im Rumpf durch den aktuellen Parameter ersetzt werden.

Beispiel

$$((\lambda\ x.\ (+\ x\ 1))\ 4) \overset{\beta}{\rightarrow} (+\ 4\ 1)$$

Das bedeutet: Die Abstraktion $(\lambda\ x.\ (+\ x\ 1))$ angewendet auf 4 führt zum Ausdruck $(+\ 4\ 1)$.
Diese Operation bezeichnen wir als *β-Reduktion* mit der Notation „$\overset{\beta}{\rightarrow}$".
Wir wollen diese Notation auch in umgekehrter Richtung zulassen:

$$(+\ 4\ 1) \overset{\beta}{\leftarrow} ((\lambda\ x.\ +\ x\ 1)\ 4)$$

Wir nennen die umgekehrte Operation *β-Abstraktion* und bezeichnen sie mit „$\overset{\beta}{\leftarrow}$".
Der Begriff *β-Konversion* bedeutet β-Reduktion oder β-Abstraktion und wird mit „$\overset{\beta}{\leftrightarrow}$" bezeichnet.

Alpha-Konversion

Die α-Konversion gibt uns die Möglichkeit der Umbenennung der gebundenen Variablen eines Ausdrucks. Mit der α-Konversion werden formale Parameter einer Lambda-Abstraktion umbenannt. Der neue Name eines formalen Parameters darf nicht im Rumpf der Lambda-Abstraktion vorkommen.

Beispiel

$$\lambda x.x \overset{\alpha}{\leftrightarrow} \lambda y.y$$

Eta-Konversion

Die Regel der η-Konversion lautet

$$(\lambda x.(F\ x)) \overset{\eta}{\leftrightarrow} F$$

wobei x in F nicht frei vorkommen darf.

Diese Konversion ist wie die β-Konversion in beide Richtungen anwendbar, wobei sie je nach Richtung η-Abstraktion bzw. η-Reduktion genannt wird.

2.1.3 Erweiterter Lambda-Kalkül

In der Theorie der Implementierung funktionaler Programmiersprachen wird oft eine erweiterte Version des Lambda-Kalküls als Quellsprache verwendet. Die Syntax des erweiterten Lambda-Kalküls ist der Abbildung 2.2 zu entnehmen.

Die Erweiterungen (**let**, **letrec**, **if**) lassen sich mit Hilfe der Basisausdrücke aus der Abbildung 2.1 ausdrücken.

Eine **let**-Bindung der Form

$$\begin{array}{ll} \textbf{let} & a_1 = e_1 \\ & a_2 = e_2 \\ \ldots & \\ & a_n = e_n \\ \textbf{in} & e \end{array}$$

bedeutet die Bindung der Variablen a_1 ... a_n an die Ausdrücke e_1 ... e_n. Die Variablenbindungen haben den gesamten Ausdruck e als Gültigkeitsbereich.

Die **letrec**-Definition erlaubt rekursive Definitionen von Funktionen. Dabei können also Funktionen vereinbart werden, die sich gegenseitig aufrufen.

Abbildung 2.2: Grammatik des erweiterten Lambda-Ausdrucks

<*exp*>::=<*constant*>	Konstanten
| <*variable*>	Variablen
| (<*exp*> <*exp*>)	Funktionsapplikationen
| λ <variable>.<*exp*>	Lambda-Abstraktion
| **let** <*binds*> **in** <*exp*>	Variablenbindungen
| **letrec** <*binds*> **in** <*exp*>	rekursive Definitionen
| **if** <*exp*> **then** <*exp*> **else** <*exp*>	bedingte Ausdrücke
<*binds*> ::= <*bind*> {; <*bind*> }	
<*bind*> ::= <*variable*> = <*exp*>	

Eine auf diese Art eingeführte **letrec**-Definition entspricht zwei unabhängigen Konzepten in prozeduralen Sprachen: den Funktionsdefinitionen (mit oder ohne *forward*-Definitionen) und den Zuweisungen.

Zur technischen Vereinfachung werden weitere alternative Notationen zugelassen. Mit

$$\lambda\ x_1\ \ldots\ x_n.e$$

bezeichnen wir die Verschachtelung von Lambda-Abstraktionen (einstellige Funktionen) des Lambda-Kalküls:

$$\lambda\ x_1.\ \ldots\ \lambda\ x_n.e.$$

Darüber hinaus wird die Lambda-Abstraktion

$$\lambda\ x_1\ \ldots\ x_n.e$$

folgendermaßen bezeichnet:

$$(LAMBDA\ (\ x_1\ \ldots\ x_n)\ e)$$

Außerdem wird eine (an dieser Stelle nicht näher bezeichnete) Menge von Standardfunktionen für die Arithmetik bzw. Listenverarbeitung vorausgesetzt.

In den folgenden Abschnitten wenden wir uns nun den Implementationstechniken funktionaler Sprachen zu.

2.2 Interpreter-basierte Implementationen

Diese Implementationstechnik beschreiben wir nur kurz, weil sie heute keine größere Bedeutung in der Implementation funktionaler Sprachen hat. Die Interpreter-basierte Implementationstechnik für funktionale Programmiersprachen wurde u.a. in [McC60] untersucht und beschrieben.

Ein Interpreter kann intuitiv als ein System für die Interpretation des funktionalen Programms, ohne ein größeres Übersetzungsverfahren anzuwenden, beschrieben werden. Das Programm wird meistens in eine Zwischensprache übersetzt und bis zu Normalform (*WHNF*) reduziert. Die Zwischensprache entspricht einem erweiterten Lambda-Ausdruck. Nach der syntaktischen Analyse des Programms erfolgt die Übersetzung in die Zwischensprache. Das in die Zwischensprache übersetzte funktionale Programm wird abstrakt im Speicher repräsentiert. Mit Hilfe zusätzlicher globaler Strukturen (wie zum Beispiel der *Environment*-Umgebung, wo die Werte aller Variablen gelagert werden) wird das Programm ausgewertet.

2.3 *Stack*-basierte Implementationen: Die abstrakte *SECD*-Maschine

In diesem Abschnit stellen wir die Semantik und die Konstruktion der *SECD*-Maschine vor. Etwas mehr über diese abstrakte Maschine kann man in [Lan64] bzw. [FH88] finden. Die *SECD*-Maschine wurde 1964 von Landin definiert [Lan64]. Die Evaluation des funktionalen Programms, die mit der Maschine durchgeführt wird, ist Umgebungs-basiert (*environment*). Das heißt, daß jede Variable ihre entsprechende *Environment*-Menge hat. Aufgrund dessen kann in jedem Moment der Evaluation auf den aktuellen Wert jeder Variablen verwiesen werden.

2.3.1 Grundzüge der *SECD*-Maschine

Der Zustand der Maschine ist durch ein 4-Tupel beschrieben: (*S, E, C, D*). *S, E, C* und *D* sind dabei die Komponenten der Maschine. Die vier Komponenten der Maschine nennen wir Register. Die Manipulation der Maschinenkomponenten ist die eigentliche Evaluation des funktionalen Programms. Die Komponenten haben folgende Bedeutung:

S: *Stack*. Die Struktur hat mehrfache Bedeutung für die Maschine: Evaluation der Ausdrücke, Speicherung der *closures*, Speicherung der return-Adressen usw.

E: Das Environment - eine Struktur, in der den Variablen ihnen entsprechende Objekte zugeordnet werden.

C: Folge von Maschineninstruktionen.

D: *Dump*. Hier wird der vorherige Zustand der Maschine gespeichert. Das ist bei dem *return* von Funktionsaufrufen notwendig.

Die Arbeit der Maschine wird mit Hilfe der Transitionsfunktion (*state transition function*) beschrieben. Das ist eine Funktion, die Manipulationen mit den Maschinenkomponenten in Abhängigkeit von dem z. Z. gültigen Zustand beschreibt. Ein Schritt der Programmevaluation wird folgendermaßen beschrieben:

$$(s,\ e,\ c,\ d) \rightarrow (s',\ e',\ c',\ d')$$

Im nächsten Schritt definieren wir die Menge der Instruktionen der abstrakten Maschine:

LD	Wert laden
LDC	Konstante laden
LDF	Funktion laden
AP	Funktionsapplikation laden
RTN	Ende der Funktion
DUM	ein leeres Environment erzeugen
RAP	rekursive Funktionsapplikation
SEL	bedingte Anweisung
JOIN	Reinstanzierung der Codesequenz vom *Dump*
CAR	Nimmt den Listenkopf der Liste vom *Stack*
CDR	Nimmt den Listenrest der Liste vom *Stack*
ATOM	*atom*-Prädikat auf oberstes *Stack*-Element anwenden
CONS	*cons*-Zelle aus den beiden obersten *Stack*-Elementen erzeugen
EQ	*eq*-Operator auf die beiden obersten *Stack*-Elemente anwenden
STOP	Ende der Programmabarbeitung

Nun folgen die arithmetischen binären Operatoren. Jeder nimmt zwei Elemente vom *Stack* und führt die entsprechende arithmetische Operation durch:

ADD
SUB
MUL
DIV
REM
LEQ

Mit der Notation

$x.S$

bezeichnen wir einen Stack mit dem obersten Element x und dem Rest S.

Mit | bezeichnen wir die Verkettung zweier Codesequenzen.
Wir benötigen noch die zweistellige Hilfsfunktion *rplaca*: *rplaca*(a, b) ersetzt den Kopf der Liste a durch b.

Für jedes funktionale Programm wird eine Liste der Namen erzeugt. Die Position eines Namens in der Liste der Namen entspricht der Position des Namens im funktionalen Quellprogramm: Wenn sich beispielsweise ein neuer Name in einem neuen Programmblock befindet, wird er in einer neuen Unterliste der Liste der Namen abgelegt. Aufgrund dessen hat jeder Name einen ihm entsprechenden Index: ($a.b$). Dabei ist a der Index der Unterliste der Liste der Namen und b der Index des Namens in der Unterliste.

Für ein funktionales Programm e und eine ihm entsprechende Liste der Namen n bezeichnen wir mit

$C_{secd}[e,\ n]$

den Maschinencode, der das Ergebnis der Übersetzung in den *SECD*-Code vom Ausdruck e ist.

2.3.2 Transformationsregeln für die Übersetzung einer einfachen funktionalen Sprache in den *SECD*-Code

Sei unsere funktionale Quellsprache ein erweiterter (*syntactic sugared*) Lambda-Kalkül mit *if*- *let*- und *letrec*-Konstruktionen (Abbildung 2.2) und Standardfunktionen für die Arithmetik und Listenverarbeitung.

Die nachfolgenden Regeln wurden in [Hen80] beschrieben:

$C_{secd}[(ADD\ e_1\ e_2),\ n] \rightarrow C_{secd}[e_1,\ n] \mid C_{secd}[\ e_2,\ n] \mid$ (ADD)

$C_{secd}[(SUB\ e_1\ e_2),\ n] \rightarrow C_{secd}[e_1,\ n] \mid C_{secd}[\ e_2,\ n] \mid$ (SUB)

$C_{secd}[(MUL\ e_1\ e_2),\ n] \rightarrow C_{secd}[e_1,\ n] \mid C_{secd}[e_2,\ n] \mid$ (MUL)

$C_{secd}[(DIV\ e_1\ e_2),\ n] \rightarrow C_{secd}[e_1,\ n] \mid C_{secd}[e_2,\ n] \mid$ (DIV)

$C_{secd}[(REM\ e_1\ e_2),\ n] \rightarrow C_{secd}[e_1,\ n] \mid C_{secd}[e_2,\ n] \mid$ (REM)

$C_{secd}[(EQ\ e_1\ e_2),\ n] \rightarrow C_{secd}[e_1,\ n] \mid C_{secd}[e_2,\ n] \mid$ (EQ)

$$C_{secd}[(LEQ\ e_1\ e_2),\ n] \rightarrow C_{secd}[e_1,\ n] \mid C_{secd}[e_2,\ n] \mid (LEQ)$$

$$C_{secd}[(CAR\ e),\ n] \rightarrow C_{secd}[e,\ n] \mid (CAR)$$

$$C_{secd}[(CDR\ e),\ n] \rightarrow C_{secd}[e,\ n] \mid (\text{CDR})$$

$$C_{secd}[(CONS\ e_1\ e_2),\ n] \rightarrow C_{secd}[e_2,\ n] \mid C_{secd}[e_1,\ n] \mid (\text{CONS})$$

$$C_{secd}[(ATOM\ e),\ n] \rightarrow C_{secd}[e,\ n] \mid (\text{ATOM})$$

$$C_{secd}[(IF\ e_1\ e_2\ e_3),\ n] \rightarrow C_{secd}[e_1,\ n] \mid (SEL\ C_{secd}[e_2,\ n] \mid (JOIN)\ C_{secd}[e_3,\ n] \mid (JOIN))$$

$$C_{secd}[(LAMBDA\ (x_1\ \ldots\ x_k)\ e),\ n\ ] \rightarrow (LDF\ C_{secd}[e, ((x_1\ \ldots\ x_k).n)] \mid (RTN))$$

$((x_1\ \ldots\ x_k).n)$ bedeutet die Erweiterung der Liste der Namen n um die Liste $(x_1\ \ldots\ x_k)$.

$$C_{secd}[(e\ e_1\ \ldots\ e_k),\ n] \rightarrow (LDC\ NIL) \mid C_{secd}[(e_k,\ n] \mid (CONS) \mid \ldots \mid C_{secd}[e_1,\ n] \mid (CONS) \mid C_{secd}[e,\ n] \mid (AP)$$

$$C_{secd}[\begin{pmatrix} \textbf{let} & x_1 = e_1 \\ & x_2 = e_2 \\ \ldots & \\ & x_k = e_k \\ \textbf{in} & e \end{pmatrix}, n] \rightarrow$$

$$\rightarrow (LDC\ NIL) \mid C_{secd}[e_k,\ n] \mid (CONS) \mid \ldots \mid C_{secd}[e_1,\ n] \mid (CONS) \mid (LDF\ C_{secd}[e,\ m] \mid (RTN)\ AP)$$

wobei m = $((x_1\ \ldots\ x_k).n)$

$$C_{secd}[\begin{pmatrix} \textbf{letrec} & x_1 = e_1 \\ & x_2 = e_2 \\ \ldots & \\ & x_k = e_k \\ \textbf{in} & e \end{pmatrix}, n] \rightarrow$$

$$\rightarrow (DUM\ LDC\ NIL) \mid C_{secd}[e_k,\ m] \mid (CONS) \mid \ldots \mid C_{secd}[e_1,\ m] \mid (CONS) \mid (LDF\ C_{secd}[e,\ m] \mid (RTN)\ RAP)$$

wobei m = $((x_1\ \ldots\ x_k).n)$

Die Transformation der Konstruktionen des Lambda-Kalküls wird durch ein rekursives Regelsystem beschrieben. Die einzelnen generierten Codeteile werden ver-

kettet. In der ersten Regel ergibt sich beispielsweise eine Verkettung aus drei Bestandteilen: Die Ergebnisse der Transformation der Parameter ($C_{secd}[e_1,\ n]$, $C_{secd}[e_2,\ n]$) werden verkettet mit der Operation ADD.

2.3.3 Operationelle Semantik des *SECD*-Codes: *Transitionen*

Die nachfolgend beschriebene operationelle Semantik stammt aus [Hen80]. Wir geben die *Transitionen* der Maschine für jeden Maschinenbefehl an. Das sind die Regeln, die die Änderungen der Maschinenkomponenten in Abhängigkeit von den Maschinenbefehlen bzw. Zuständen definieren. Aus dem System der *Transitionen* kann die operationelle Semantik der Maschine ermittelt werden.

Mit $Z_1 \rightarrow Z_2$ bezeichnen wir, daß die Maschine vom Zustand Z_1 in den Zustand Z_2 überführt wird. Ein Zustand Z wird durch die vier Komponenten (S, E, C, D) der Maschine definiert.

Beginnen wir mit den arithmetischen Operationen:

$$((a\ b.\mathrm{s}),\ e,\ (ADD.c),\ \mathrm{d}) \rightarrow (((b{+}a).\mathrm{s}),\ e,\ c,\ d)$$

Vor der Anwendung der Operation ADD (+) waren die Argumente a und b auf dem *Stack* (S). Auf dem *Stack* wird die Summe der zwei Argumente abgelegt und die Operation *ADD* wird aus dem Register C gestrichen. Auf die gleiche Weise werden die anderen arithmetischen Operatoren behandelt.
Nun folgen die Transitionsregeln für die Listenoperationen:

$$((a\ b.\mathrm{s}),\ e,\ (CONS.\mathrm{c}),\ \mathrm{d}) \rightarrow ((a.b).\mathrm{s},\ e,\ c,\ d)$$

Mit $(a.b)$ haben wir die Konstruktion eines S-Ausdrucks, dessen Kopf a und Rest b ist, bezeichnet.

$$(((a.b).s),\ e,\ (CAR.c),\ d) \rightarrow (a.s,\ e,\ c,\ d)$$

$$(((a.b).s),\ \mathrm{e},\ (CDR.c),\ d) \rightarrow ((b.s),\ e,\ c,\ d)$$

$$((a.s),\ e,\ (ATOM.c),\ d) \rightarrow ((t.s),\ e,\ c,\ d)$$

wobei folgendes gilt: t=T, falls a ein Atom ist, ansonsten t=F.

Die anderen Transitionen sind:

$$(x.s,\ \mathrm{e},\ (SEL\ c_t\ c_f.c\ ),\ d) \rightarrow (s,\ e,\ c_x,\ (c.d))$$

wobei:

$$c_x = \begin{cases} c_t, & falls\ x\ =\ T; \\ \\ c_f, & ansonsten. \end{cases}$$

$$(s,\ e,\ (JOIN),\ (c.d)) \rightarrow (s,\ e,\ c,\ d)$$

Nach dem Befehl *SEL* wird das *C*-Register auf dem *Dump* (Register *D*) abgelegt. Mit *JOIN* wird der Code des Registers *C* wieder aktualisiert: Das gespeicherte Register *C* wird von *Dump* genommen.

$$(s,\ e,\ (LD\ i.c),\ \mathrm{d}) \rightarrow ((x.s),\ e,\ c,\ d)$$

wobei x der der Position i im Environment e entsprechende Wert ist. Das ist ein Verweis auf den Wert einer Variablen.

$$(s,\ e,\ (LDF\ c'.c),\ d) \rightarrow (((c'.e).s),\ e,\ c,\ d)$$

Mit *LDF* wird eine Funktionsapplikation vorbereitet: Der Rumpf der Funktion und ihr entsprechendes Environment werden auf das Register *S* gelegt.

$$(((c'.e')v.s),\ e,\ (AP.c),\ \mathrm{d}) \rightarrow (NIL,\ (v.e'),\ c',\ (s\ e\ c.d))$$

Das ist eine Funktionsapplikation. Auf dem Register *S* befindet sich der Rumpf der aufzurufenden Funktion mit dem entsprechenden Environment (siehe die Transition von *LDF*) und den Argumenten. Die Register *S*, *E* und *C* werden im *Dump* gespeichert. Das neue *S*-Register ist leer, das neue Environment ist das mit Argumenten der Funktionsapplikation erweiterte aktuelle Environment der Funktion (e') und die neue zu evaluierende Codesequenz (Register *C*) ist der Code der Funktion (c'). Mit NIL wird ein leeres Register bezeichnet.

$$((x),\ e',\ (RTN),\ (s\ e\ c.d)) \rightarrow ((x.s),\ e,\ c,\ d)$$

Mit RTN (*return*) wird der vor einer Funktionsapplikation geltende Zustand der Maschine wiederhergestellt.

$$(s,\ e,\ (DUM.c),\ d) \rightarrow (s,\ (\Omega.e),\ c,\ d)$$

Ω ist ein Spezialzeichen, mit dessen Hilfe Zugriffe zu einem Environment vermieden werden. Der Versuch, auf einen Wert in *environment* e zu verweisen, wird erfolglos bleiben, solange Ω in e liegt. Das wird bei der Bearbeitung der rekursiven Definitionen verwendet (*letrec*). Die Behandlung von Ω wird in der folgenden Transition definiert.

$$(((c'.e')\ v.s),\ (\Omega.e),\ (RAP.\mathrm{c}),\ d) \rightarrow (NIL,\ rplaca(e',\ v),\ c',\ (s\ e\ c.d))$$

Das ist eine Applikation einer rekursiven Funktion. Der Befehl ist ähnlich dem Befehl *AP*. Durch den Befehl *RAP* wird Ω mit Hilfe von *rplaca* gestrichen.

Beispiel:

Übersetzen wir die folgende Funktionsapplikation:

$$((\mathrm{LAMBDA}(x\ y)\ (\mathrm{ADD}\ x\ y))\ 2\ 3)$$

Nach dem o.g. Übersetzungsalgorithmus wird das funktionale Programm in den *SECD*-Code übersetzt:

$$C_{secd}[((\ \mathrm{LAMBDA}(x\ y)\ (\mathrm{ADD}\ x\ y))\ 2\ 3),\ ((x\ y))] \rightarrow \\ LDC\ NIL \mid LDC\ 3 \mid (CONS) \mid LDC\ 2 \mid (CONS) \mid \\ (LDF\ LD\ (0\ 0) \mid LD\ (0\ 1) \mid (ADD) \mid (RTN)) \mid (AP)$$

Die Übersetzung erfolgt nach den Transformationsregeln für die Funktionsapplikation, die Konstanten und die Lambda-Abstraktion.

Die Evaluation des Maschinencodes mit Hilfe der Zustände der Maschinenkomponenten ist in der Abbildung 2.3 dargestellt.

Wenn der Maschinenbefehl *STOP* abgearbeitet wird, bedeutet das, daß der Ablauf des Programms beendet ist und das Ergebnis der Evaluation oben auf dem *Stack* liegt.

Die Maschine evaluiert das funktionale Programm strikt. Die Argumente einer Funktion werden vor der Bearbeitung der Funktion ausgewertet (*call by value*). In der Literatur wurden auch nicht-strikte Versionen der Maschine betrachtet. Die nicht-strikte Evaluation basierte auf der Kontrolle der Funktionsapplikation. Die Argumente der aufgerufenen Funktion werden nicht ausgewertet, sondern ihre Evaluation wird verzögert. Wenn die nicht ausgewerteten Argumente für die weitere Bearbeitung des funktionalen Programms notwendig sind, wird deren Evaluation erzwungen.

2.4 Graph-Reduktion: Abstrakte *SK*-Maschine

Bei der *SECD*-Maschine war der Code der abstrakten Maschine einem konkreten Assemblercode sehr ähnlich. Der Programmablauf fordert ständige Informationen über die Variablenbindungen, die mit Hilfe des Environments gespeichert werden. Die nicht-strikten Versionen der *SECD*-Maschine basierten auf einem speziellen Mecha-

Abbildung 2.3: Evaluation des SECD-Codes

S	*E*	*C*	*D*
	NIL	*LDC NIL* | *LDC* 3 | (*CONS*) | *LDC* 2 | (*CONS*) | (*LDF LD*(0 0) | *LD* (0 1) | (*ADD*) | (*RTN*)) | (*AP*) | *STOP*	NIL
()	NIL	*LDC* 3 | (*CONS*) | *LDC* 2 | (*CONS*) | (*LDF LD*(0 0) | *LD* (0 1) | (*ADD*) | (*RTN*)) | (*AP*) | *STOP*	NIL
3 ()	NIL	(*CONS*) | *LDC* 2 | (*CONS*) | (*LDF LD*(0 0) | *LD* (0 1) | (*ADD*) | (*RTN*)) | (*AP*) | *STOP*	NIL
(3)	NIL	*LDC* 2 | (*CONS*) | (*LDF LD*(0 0) | *LD* (0 1) | (*ADD*) | (*RTN*)) | (*AP*) | *STOP*	NIL
2 (3)	NIL	(*CONS*) | (*LDF LD*(0 0) | *LD* (0 1) | (*ADD*) | (*RTN*)) | (*AP*) | *STOP*	NIL
(2 3)	NIL	(*LDF LD*(0 0) | *LD* (0 1) | (*ADD*) | (*RTN*)) | (*AP*) | *STOP*	
(((*LD*(0 0) | *LD* (0 1) | (*ADD*) | (*RTN*)) . NIL) ((2 3)))	NIL	(*AP*) | *STOP*	NIL
NIL	((2 3))	*LD* (0 0) | *LD* (0 1) | (*ADD*) | (*RTN*)	NIL ((2 3)) *STOP* NIL
2	((2 3))	*LD* (0 1) | (*ADD*) | (*RTN*)	NIL ((2 3)) *STOP* NIL
3 2	((2 3))	(*ADD*) | (*RTN*)	NIL ((2 3)) *STOP* NIL
5	((2 3))	(*RTN*)	NIL ((2 3)) *STOP* NIL
5 NIL	((2 3))	*STOP*	NIL

nismus für die *lazy evaluation*. Die Nicht-Striktheit ist auf eine sehr ,,unnatürliche" Art und Weise realisiert worden. In diesem Abschnitt wird eine ,,natürliche" Implementationstechnik nicht-strikter funktionaler Programmiersprachen vorgestellt.

2.4.1 Grundzüge der SK-Graph-Reduktion

Die Technik der Graph-Reduktion wurde von Wadsworth [Wad71] eingeführt.

Bei diesem Ansatz werden die Variablenbindungen mit Pointern dargestellt (*sharing*), und der Ausdrucksbaum bekommt die Form eines Graphen. Durch Graphen repräsentierte Bindungen haben folgende Vorteile:

- Jede Variablenbindung kann mit Hilfe einer Kante im Graphen (ein *Pointer*) dargestellt werden, so daß keine zusätzlichen Datenstrukturen erforderlich sind.

- Dieses Modell ist für die *call-by-need*-Evaluation sehr geeignet.

Die Technik der SK-Graph-Reduktion beruht dabei auf dem Begriff des Kombinators. Ein *Kombinator* ist ein Lambda-Ausdruck, der keine freien Variablen hat [Bar84].

So sind beispielsweise folgende Lambda-Ausdrücke Kombinatoren:

$\lambda f\ g\ x.\ f\ x\ (g\ x)$
$\lambda c\ x.c$
$\lambda x.x$

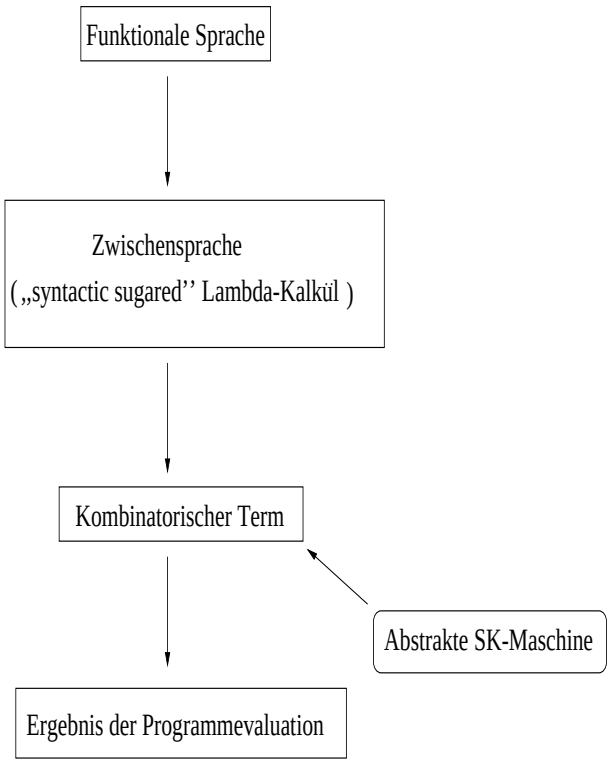
Die o.g. Kombinatoren heißen in der aufgeführten Reihenfolge *S*-, *K*- bzw. *I*-Kombinator. Alle Kombinatoren können durch *S*- und *K*-Kombinatoren dargestellt werden (z. B. $I = S\ K\ K$).

Das allgemeine Schema der durch SK-Kombinatoren realisierten Implementation einer funktionalen Sprache finden wir in Abbildung 2.4.

Dabei wird die funktionale Sprache in kombinatorische Terme transformiert. Ein kombinatorischer Term wird dann durch die abstrakte SK-Maschine verarbeitet. Eine Reihe von Reduktionsschritten wird solange ausgeführt, bis der kombinatorische Term schwache Kopfnormalform (*WHNF*) erreicht [Jon87].

Im folgenden werden wir den Algorithmus der Übersetzung einer einfachen funktionalen Sprache (erweiterter Lambda-Ausdruck) in einen kombinatorischen Term und die Reduktionsregeln der Kombinatoren angeben [Jon87].

Abbildung 2.4: Globales Schema der auf SK-Kombinatoren besierenden Implementation einer funktionalen Programmiersprache



2.4.2 Übersetzungsregeln

Sei unsere Quellsprache ein Lambda-Ausdruck wie in Abbildung 2.2 definiert. Außerdem nehmen wir zusätzlich definierte arithmetische Operationen an und setzen eine nicht-strikte (*call-by-need*) Semantik voraus.

An dieser Stelle wollen wir die Grundidee der Übersetzung einer funktionalen Sprache in kombinatorische Terme darstellen, um die Grundzüge der abstrakten SK-Maschine deutlicher beschreiben zu können.

Bezeichnen wir die Übersetzungsfunktion, die einen Lambda-Ausdruck in einen SK-kombinatorischen Term übersetzt, mit **C**. Die Übersetzung eines funktionalen Ausdrucks e führt dann zu seinem kombinatorischen Äquivalent $\mathbf{C}\ [e]$.

Für die Definition der Übersetzungsfunktion **C** ist es notwendig, eine Hilfsfunktion **A** zum Abstrahieren einer Variablen x aus einem funktionalen Ausdruck *exp* zu definieren:

$$\mathbf{A}\ x\ [\ exp\ ]$$

Die beiden Funktionen **A** und **C** liefern Lambda-Ausdrücke, die als Argumente der Funktion **C** bzw. als zweites Argument der Funktion **A** auftreten können. In den folgenden Übersetzungsregeln haben die aufgetretenen Bezeichner folgende Bedeutung:

$e,\ e_1,\ e_2$	beliebige Ausdrücke der funktionalen Quellsprache
$f_1,\ f_2$	Ausdrücke ohne innere λ-Abstraktionen
x	Variablen
cv	Konstanten oder Variablen

Geben wir nun die Definition der Übersetzungsfunktion an ([Jon87]):

$$\mathbf{C}[(e_1\ e_2)\ ] = \mathbf{C}[e_1]\ \mathbf{C}[e_2]$$

$$\mathbf{C}[\lambda x.e] = \mathbf{A}\ x\ [\mathbf{C}[e]]$$

$$\mathbf{C}[cv] = cv$$

Die Abstraktionsfunktion wird folgendermaßen definiert:

$$\mathbf{A}\ x\ [(f_1\ f_2)] = S\ (\mathbf{A}\ x\ [f_1])\ (\mathbf{A}\ x\ [f_2])$$

$$\mathbf{A}\ x\ [x] = I$$

$$\mathbf{A}\ x\ [\ cv] = K\ cv$$

Das Übersetzungsverfahren für **let**- und **letrec**-Konstruktionen geben wir an

dieser Stelle nicht an. Man kann es z. B. in [Rev88, Dil88] finden.

2.4.3 Reduktionsregeln der Kombinatoren

Jeder Kombinator ist eine Lambda-Abstraktion, und aufgrund dessen kann eine ihm entsprechende Reduktionsregel definiert werden. So lauten beispielsweise die Reduktionsregeln der Kombinatoren *S, K* und *I* folgendermaßen:

$$S\ f\ g\ x \rightarrow f\ x\ (g\ x)$$

$$K\ c\ x \rightarrow c$$

$$I\ x \rightarrow x$$

Beispiel

Übersetzen wir den Ausdruck: $((\lambda x.\ (+\ x\ x))\ 5)$

$$\begin{array}{l}
\mathbf{C}[\ ((\lambda x.\ (+\ x\ x))\ 5)] = \\
\mathbf{C}[\ (\lambda x.\ (+\ x\ x))]\ \mathbf{C}[5] = \\
\mathbf{A}\ x\ [\mathbf{C}[(+\ x\ x)]]\ 5 = \\
\mathbf{A}\ x\ [\ (+\ x\ x)]\ 5 = \\
S\ (\mathbf{A}\ x\ [(+\ x)])\ (\mathbf{A}\ x\ [x])\ 5 = \\
S\ (S\ (\mathbf{A}\ x\ [+])\ (\mathbf{A}\ x\ [x]))\ I\ 5 = \\
S\ (S\ (K\ +)\ I)\ I\ 5
\end{array}$$

Reduzieren wir nun den kombinatorischen Term nach den angegebenen Reduktionsregeln:

$$\begin{array}{l}
S\ (S\ (K\ +)\ I)\ I\ 5 \rightarrow \\
S\ (K\ +)\ I\ 5\ (I\ 5) \rightarrow \\
K\ +\ 5\ (I\ 5)\ (I\ 5) \rightarrow \\
+\ (I\ 5)\ (I\ 5) \rightarrow \\
+\ 5\ (I\ 5) \rightarrow \\
+\ 5\ 5 \rightarrow \\
10
\end{array}$$

Die Funktion ,,+" ist eine Standardfunktion, und ihre Definition entspricht der Definition der entsprechenden arithmetischen Funktion.

2.4.4 Optimierung der abstrakten SK-Maschine durch die Erweiterung der Kombinatorenmenge

Wir haben den Algorithmus mit drei Kombinatoren dargestellt. In der Praxis wird eine größere Menge von Kombinatoren eingeführt. Die Einführung eines neuen Kombinators ist eine Sache der Optimierung. Mit nur drei Kombinatoren S, K und I wäre der Maschinencode zu groß und die Implementation nicht effizient.
Die neuen Kombinatoren können durch die Optimierungsfunktion **Opt** generiert werden. Die Optimierungsfunktion wird durch die Abstraktionsfunktion in den Übersetzungsalgorithmus eingeführt ([Jon87]):

$$\mathbf{A}\ x\ [(f_1\ f_2)] = \mathbf{Opt}[\ S\ (\mathbf{A}\ x\ [f_1])\ (\mathbf{A}\ x\ [f_2])]$$

$$\mathbf{A}\ x\ [x] = I$$

$$\mathbf{A}\ x[\ cv] = K\ cv$$

In der Implementation funktionaler Programmiersprachen ist die Turner-Menge der Kombinatoren [Tur79] besonders effizient:

S, K, I, B, C, S', C', B'

wobei

$$\begin{array}{ll}
\mathbf{Opt}[S(K\ p)\ q] & = B\ p\ q \\
\mathbf{Opt}[S\ p\ (K\ q)] & = C\ p\ q \\
\mathbf{Opt}[S\ (B\ p\ q)\ r] & = S'\ p\ q\ r \\
\mathbf{Opt}[S\ (B\ p\ q)\ K\ r] & = C'\ p\ q\ r \\
\mathbf{Opt}[B\ (p\ q)\ r] & = B'\ p\ q\ r
\end{array}$$

Die Reduktionsregeln der o.g. Kombinatoren lauten:

$$\begin{array}{ll}
B\ f\ g\ x & \rightarrow f\ (g\ x) \\
C\ f\ g\ x & \rightarrow f\ x\ g \\
S'\ c\ f\ g\ x & \rightarrow c\ (f\ x)\ (g\ x) \\
C'\ c\ f\ g\ x & \rightarrow c\ (f\ x)\ g \\
B'\ c\ f\ g\ x & \rightarrow c\ f\ \ (g\ x)
\end{array}$$

Bei der Implementation einer funktionalen Programmiersprache werden auch alle Standardfunktionen als Kombinatoren definiert.

An dieser Stelle haben wir nur die grundlegenden Prinzipien der Übersetzung eines Lambda-Ausdrucks in einen kombinatorischen Term angegeben. Mehr über die Übersetzung einer funktionalen Sprache in kombinatorische Terme kann man z.

B. in [Dil88], [Jon87] (Kapitel 16), [Rev88] (Kapitel 3 und 6) und [FH88] (Kapitel 12) finden.

2.4.5 SK-Graph-Reduktion

Wir haben die grundlegenden Prinzipien der Übersetzung eines funktionalen Programms in einen kombinatorischen Term und entsprechende Reduktionsregeln für Kombinatoren angegeben. An dieser Stelle erläutern wir den Prozeß der SK-Graph-Reduktion.

In der Abbildung 2.5 ist die innere Repräsentation eines kombinatorischen Terms dargestellt. **ROOT** ist die Wurzel des Graphen eines Ausdrucks. Die Reduktion läuft mit Hilfe eines *Stacks* ab, der nach jedem Reduktionsschritt rekonstruiert werden soll. Die Graph-Reduktion erfolgt nach dem Prinzip der Ersetzung der **ROOT** mit dem Ergebnis der Evaluation des Kombinators **Komb**.
In der Abbildung 2.6 ist die Reduktion des Ausdrucks (+ (+ 10 30) 20) dargestellt. Die Reduktionsschritte in diesem Beispiel sind:

1. Evaluation des ersten Arguments der Operation +: Rekonstruktion des Stacks für die Evaluation des Ausdrucks mit dem **ROOT**-Knoten $, Berechnung des Ausdrucks ($), Ersetzung des Knotens $ mit dem Ergebnis der Berechnung und Rekonstruktion des Stacks. Mit **Garbage** ist der Teil des Graphen bezeichnet, der für die weitere Evaluation nicht mehr nötig wird.

2. Durchführung der arithmetischen Operation + mit den Argumenten 40 und 20 und Ersetzung des **ROOT** durch das Ergebnis der Berechnung.

3. Rekonstruktion des Stacks.

In diesem Beispiel haben wir gezeigt, wie die Graph-Reduktion durchgeführt wird. Die arithmetischen Funktionen sind strikt und verlangen die Berechnung ihrer Argumente, bevor die Operation ausgeführt wird. Bei der Reduktion anderer, nichtstrikter Funktionen (Turners Kombinatoren) werden die Argumente der Funktionen nicht berechnet.

Die Implementationstechnik haben wir nur in wesentlichen Punkten vorgestellt. Vorteile derartiger Implementationen sind:

- Die Reduktionsmaschine für den kombinatorischen Term ist einfach zu implementieren.

- Die Implementationstechnik unterstützt die nicht-strikte Semantik einer Sprache[2].

[2]Es gibt eine Technik der *indirections* für die Vermeidung des mehrmaligen Berechnens eines Ausdrucks. Das ist notwendig, damit in der Manipulation mit dem Graphen ein unausgewerteter Knoten (*redex*) nicht kopiert wird (*lazy evaluation*).

Abbildung 2.5: Interne Repräsentation des SK-kombinatorischen Terms

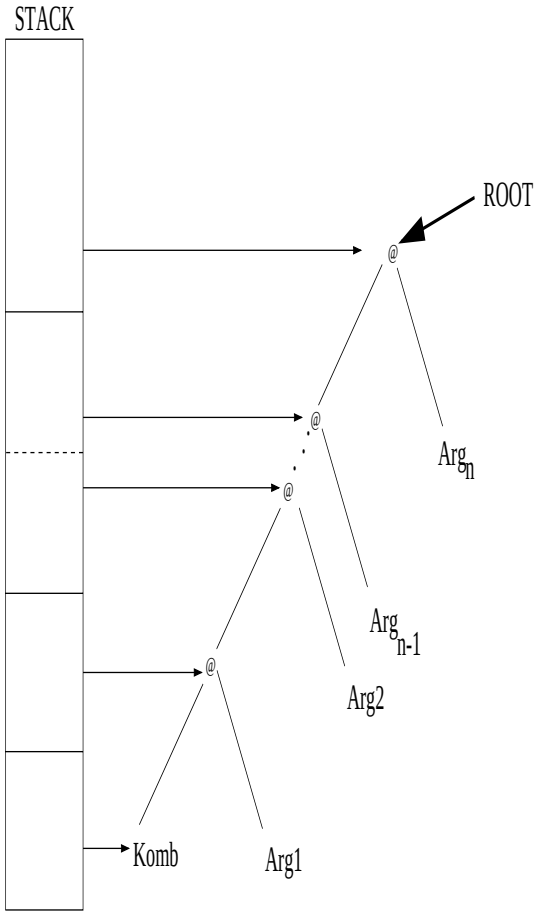
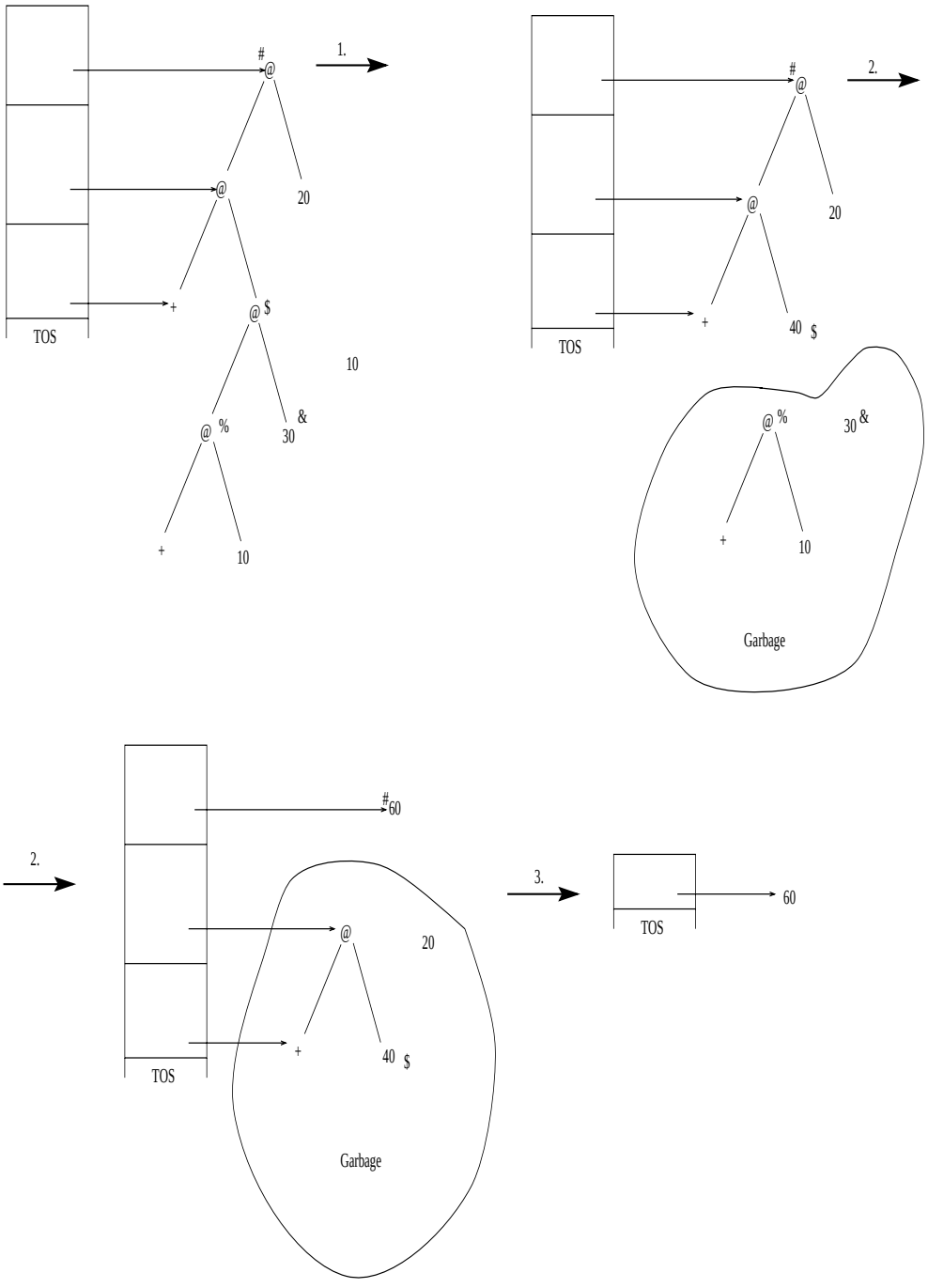


Abbildung 2.6: Reduktion des Ausdrucks (+ (+ 10 30) 20)



- Die Kombinatoren könnten auch in einer Hardwarerealisierung implementiert werden.

Nachteile dieser Implementation sind:

- Die Graph-Reduktion benötigt viel Speicher. Darüber hinaus bleiben nach einigen Reduktionen mehrere nicht mehr notwendige Knoten übrig, die im folgenden Aufruf des Garbage Collectors freigegeben werden sollen. Das vermindert die Effizienz.
- Der zu reduzierende kombinatorische Term ist manchmal recht groß.

2.5 Die abstrakte *G*-Maschine

Die abstrakte G-Maschine ist eine Technik, die sowohl auf der Stackmanipulation als auch auf der Graph-Reduktion basiert.

Die *G*-Maschine wurde am *Chalmers Institute of Technology, Göteborg, Sweden* von Johnsson und Augustsson entwickelt [Aug84, Joh84, AJ89]. Sie ist eine Mischung der zwei letztgenannten Techniken, die auf Superkombinatoren (Lambda-Ausdrücke ohne freie Variablen, wobei beliebige Unterausdrücke ebenfalls Superkombinatoren sind) basiert. Sie ist der *SK*-Maschine ähnlich, wobei bei der *G*-Maschine ausschließlich Superkombinatoren definiert werden.

Komponenten der abstrakten *G*-Maschine sind:

- *S* - Stack der Maschine
- *G* - zu reduzierender Graph
- *C* - Code der abstrakten Maschine
- *D* - *Dump*.

Die Maschine wird oft als *SGCD*-Maschine bezeichnet. Die Semantik der *G*-Maschine ist wie die Semantik der abstrakten *SECD*-Maschine durch Transitionen definiert [AJ89].

Der Compilationsprozeß für die Übersetzung einer funktionalen Programmiersprache in den G-Code besteht aus der Übersetzung des funktionalen Quellprogramms in eine Form ohne innere Lambda-Abstraktionen und freie Variablen (*Lambda-Lifting*) und der daran anschließenden Transformation in den *G*-Code. Der *G*-Code wird mit Hilfe der abstrakten Maschine evaluiert. Während des Evaluationsprozesses wird durch die Maschine die Erzeugung und die Reduktion des Graphen durchgeführt.

Die Befehle der abstarkten G-Maschine können in zwei Gruppen unterteilt werden:

1. Befehle für die Stack-Manipulation

2. Befehle für die Graph-Reduktion (Erzeugung und Evaluation des Graphen).

Für die Graph-Reduktion ist wie bei der abstrakten SK-Maschine ein Mechanismus definiert:

- Graphrepräsentation des auszuwertenden Ausdrucks, „Wirbelsäule " genannt (engl. spine), ähnlich wie bei der abstrakten SK-Maschine (Abbildung 2.5)

- Stack für die Evaluation des Graphen (engl. spine stack).

An der Universität Glasgow ist eine optimierte Version dieser Maschine entwickelt worden - die abstrakte *STG*-Maschine (*Spineless Tagless G-machine*) [PJ92]. Die Optimierung besteht in folgenden Innovationen: Alle Objekte der Maschine werden uniform durch *closures* repräsentiert. Bei solcher Behandlung der Objekte sind Tag-Felder, die Informationen über Typen enthalten, nicht mehr notwendig, und die Analyse der Tags wird bei der Evaluation nicht mehr durchgeführt (daher kommt der Name *tagless*). Der zu reduzierende Graph wird nicht auf die gleiche Art und Weise wie bei den abstrakten *G*- und *SK*-Maschinen repräsentiert. Der Programmablauf wird durch einen Stack-basierten *Continuations*-Mechanismus gesteuert. Die abstrakte *STG*-Maschine hat keine „Wirbelsäule" (*Spineless*) und keinen Stack für ihre Evaluation.

Da die Grundzüge der Evaluation der abstrakten *STG*-Maschine gleich den Grundzügen der Programmevaluation der abstrakten *G*-Maschine sind, wird die abstrakte *STG*-Maschine nur als eine Optimierung der abstrakten *G*-Maschine betrachtet.[3]

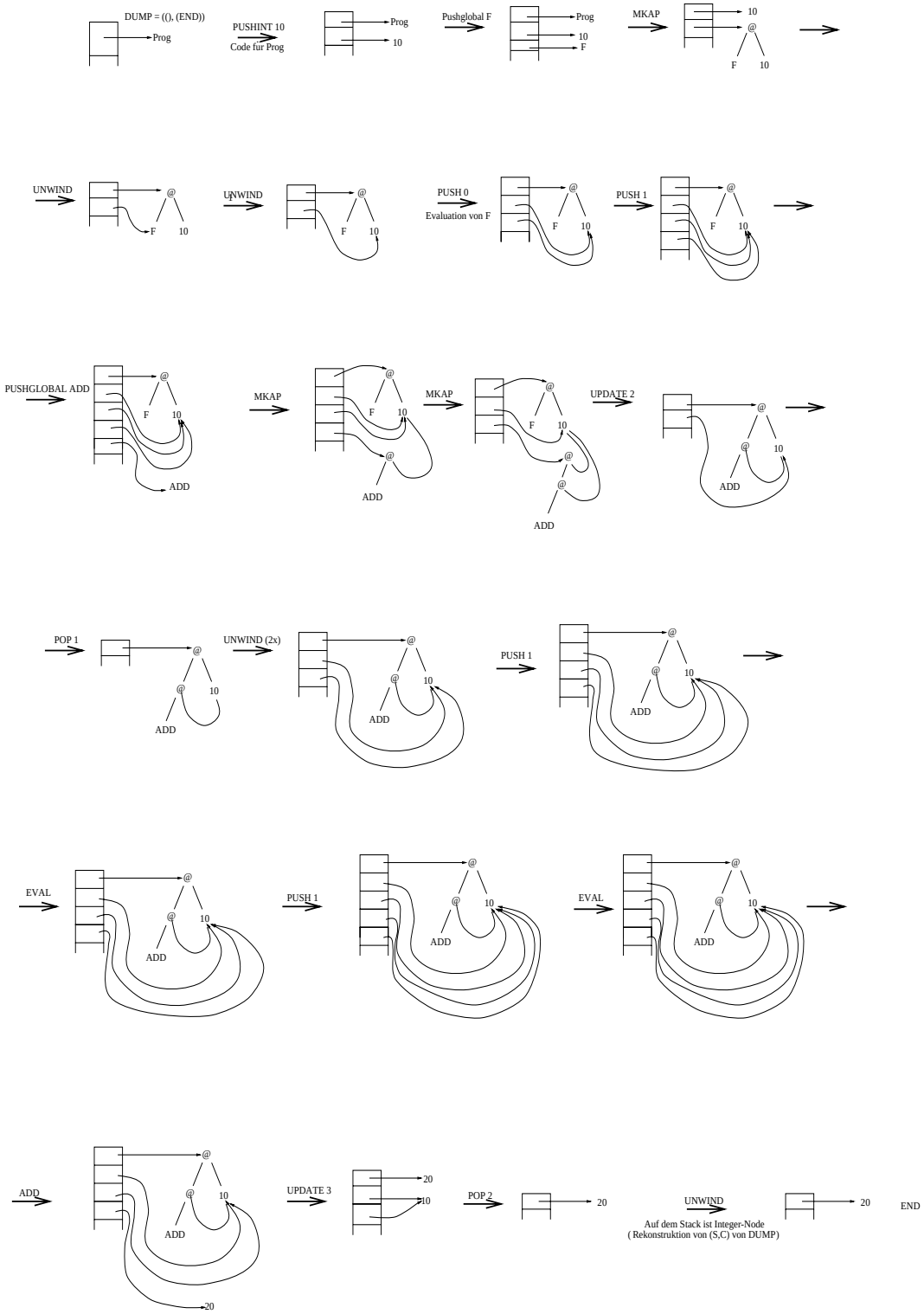
In der Abbildung 2.7 ist ein Beispiel der Evaluation des *G*-Codes dargestellt. Die Abbildung stellt die Operationen mit der G-Komponente der abstrakten Maschine (der Graph) bzw. mit dem Spine-Stack des Evaluationsmechanismus dar. Die durchgeführten Maschinenbefehle sind über den Pfeilen, die Zustandänderungen symbolisieren, bezeichnet.[4]

An dieser Stelle sollte nur eine knappe Beschreibung der abstrakten *G*-Maschine gegeben werden. Regeln für die Übersetzung eines funktionalen Programms in den *G*-Code und die operationelle Semantik der *G* Maschine kann man in [Jon87, AJ89] finden.

[3]Es gibt Beispiele, in denen die abstrakte *G*-Maschine schneller bzw. effizienter als die abstrakte *STG*-Maschine ist.

[4]Es ist nicht beabsichtigt, daß der Leser das Beispiel zur *G*-Maschine in jedem Detail versteht. Vielmehr soll das Prinzip verdeutlicht werden (Aufbau und Evaluation des Graphen durch Maschinenbefehle).

Abbildung 2.7: Beispiel der Evaluation des G-Codes



Kapitel 3

Direkte Übersetzung funktionaler in prozedurale Sprachen

In diesem Kapitel wird eine direkte Übersetzung einer funktionalen in eine prozedurale Sprache vorgestellt. Die Transformation selbst unterteilen wir in zwei Aspekte:

- Übersetzung jedes funktionalen Ausdrucks in sein prozedurales Äquivalent
- Übersetzung der Typen der funktionalen Quellsprache.

Zunächst wird die Problematik der Übersetzung, die aufgrund der Unterschiede der beiden Paradigmen auftritt, diskutiert.

Danach wird ein Algorithmus für die Übersetzung eines erweiterten Lambda-Kalküls, der unsere Quellsprache ist, in eine Zwischensprache angegeben. Die Zwischensprache hat eine prozedurale Form, die leicht in eine konkrete prozedurale Sprache zu übersetzen ist.

Nach der Definition der Transformationsfunktion folgt die Behandlung der Typen der funktionalen Quellsprache.

Als Beispiel betrachten wir die Implementation von *Scheme* durch Übersetzung nach *Modula-2* mit der dargestellten Technik in einem von uns realisierten Compiler.

Der von uns implementierte Compiler wird mit anderen, aus der Literatur bekannten direkt übersetzenden Compilern verglichen, wobei wir uns auf den *Scheme* $\rightarrow$ *C*-Compiler von Bartlett [Bar93] konzentrieren.

3.1 Problematik der Quelltexttransformationen

Der Unterschied zwischen prozeduralen und funktionalen Sprachen ist groß. In diesem Abschnitt geben wir eine Übersicht über die wichtigsten konzeptionellen Unterschiede.

Typsysteme

Betrachten wir zunächst wichtige Eigenschaften von Typsystemen in Programmiersprachen.

Wenn jedem Ausdruck einer Sprache sein Typ während der Übersetzungszeit zugeordnet werden kann, handelt es sich um ein *statisches* Typsystem. Falls die Typen erst während der Laufzeit ermittelt werden können, ist das Typsystem dann *dynamisch*.

Die Sprache *Scheme* [RE91] ist beispielsweise eine dynamisch getypte funktionale Sprache. Das Typsystem von *Haskell* [HPJWe92] ist statisch.

Die prozeduralen Sprachen werden meistens statisch typisiert, d. h. die Typen sind während der Compilation bekannt.

Eine statische Typisierung hat unter dem Aspekt der Quelltexttransformation eine Reihe von Vorteilen gegenüber der dynamischen Typisierung:

- **Einsparung von Hauptspeicherplatz durch statische Information über den Typ**
  Die meisten Quelltextcompiler funktionaler Sprachen (z. B. [Bar93, Tam93]) basieren auf der uniformen Repräsentation aller Objekte, d. h. die Datenobjekte werden in der Zielsprache einheitlich durch Pointer repräsentiert, wobei jedem Objekt sein Typ (*Tag*) zugeordnet wird. Vor den Operationen, die in der Zielsprache bestimmte Typen verlangen, werden Konvertierungsfunktionen aufgerufen, die die uniforme Repräsentation unter Auswertung des Typ-Tags in die konkrete Repräsentation eines bestimmten Typs (z. B. *Integer*) überführen. Solche Repräsentationen der Daten heißen *boxed*-Repräsentationen. Derartige Implementationen verlangen mehr Speicherplatz.

- **Größere Laufzeiteffizienz der erzeugten Programme aufgrund von Basistypen im Zielcode**
  Für die Ausführung einer einfachen arithmetischen Operation beim *Boxing* sind beispielsweise folgende Schritte erforderlich:

  1. das „Herausnehmen" der Argumente aus der *Box*
  2. die Berechnung der Operation mit den Argumenten
  3. Speicherzuweisung einer *Box* für das Ergebnis der Berechnung
  4. das Einbauen des Ergebnisses in die *Box*.

  Erzeugt eine Quelltexttransformation einen Zielcode, der direkt auf den Basistypen (einfachen Typen) der Zielsprache beruht, so müssen die betreffenden einfachen Typen nicht mehr in eine Box eingebaut werden. Dies bedeutet, daß die Schritte 1, 3 und 4 nicht mehr auszuführen sind und derartige

Programme demnach schneller laufen. Objekte mit diesem vereinfachten Ansatz werden als *unboxed*-Objekte bezeichnet. Heute sind mehrere fortgeschrittene Implementationen moderner Programmiersprachen bekannt, die *unboxed*-Datenobjekte unterstützen, wie z. B. die an der Glasgow Universität entwickelte *STG-Maschine*-basierte Implementation von Haskell, welche uneingebaute Datenobjekte auf dem Niveau der abstrakten Maschine unterstützt [PJL91].

- **Programmablauf ohne Typkontrolle**
  Wenn das Programm während der Übersetzung auf Typübereinstimmung überprüft wurde, ist es nicht nötig, während der Laufzeit die Typkontrolle durchzuführen. Dies macht das Programm effizienter.

- **Programmablauf ohne Typkonflikte**
  Nach der Typübereinstimmungkontrolle zur Übersetzungszeit sind wir sicher, daß zur Laufzeit keine Typkonflikte erscheinen werden - *well-typed programs cannot ,,go wrong"* [Mil78]. Hierdurch gestaltet sich die Fehlersuche für den Programmierer komfortabler.

Auswertungsstrategie

Bei den funktionalen Programmiersprachen werden folgende Auswertungsstrategien am häufigsten benutzt:

- *Strikte Auswertung*:
  Es werden Argumente einer Funktion berechnet bevor die Abbildungsvorschrift der Funktion angewendet wird.

- *Nicht-strikte Auswertung*:
  Die Abbildungsvorschrift der zu evaluierenden Funktion wird angewendet. Ihre Argumente werden nur berechnet, falls es nötig ist.

- *Verzögerte Auswertung*:
  Nicht-strikte Auswertung, bei der alle Ausdrücke maximal einmal ausgewertet werden.

Eine strikte funktionale Programmiersprache ist z. B. *Scheme*, und eine nicht-strikte funktionale Programmiersprache ist z. B. *Haskell*. In der Sprache *Scheme* gibt es Möglichkeiten für die Manipulation der Auswertung (Sprachelemente *delay* und *force*). Zusätzlich zur eingebauten strikten Auswertung kann damit eine nicht-strikte Auswertung Nutzer-gesteuert realisiert werden.

Da sich die heutzutage verbreiteten prozeduralen Programmiersprachen (*Pascal, Modula-2, C, Ada* usw.) im Sinne einer strikten Auswertungsstrategie verhalten, sind nicht-strikte Konzepte funktionaler Sprachen ein besonderes Problem bei der Implementation der direkten Übersetzer.

Funktionen höherer Ordnung

Wenn in einer Sprache Funktionen Objekte erster Ordnung sind, heißt das, daß sie selbst wie Datenobjekte behandelt werden können. Funktionen können sowohl Argumente anderer Funktionen als auch von anderen Funktionen gelieferte Objekte sein. Eine Funktion in den meisten Sprachen des prozeduralen Paradigmas kann nicht auf diese Art und Weise behandelt werden.

Musteranpassung (*pattern matching*)

In den modernen funktionalen Programmiersprachen werden die Benutzer-definierten Funktionen durch Musteranpassung definiert. Die Evaluationsstrategie der Ausdrücke bei der Musteranpassung ist sehr wichtig. Die Übersetzung der Musteranpassung in das prozedurale Paradigma ist problematisch.

List-Comprehensions

In modernen funktionalen Sprachen gibt es eine besondere Notation für die Definition von Listen, die dem mathematischen Formalismus der Mengen sehr ähnlich ist. Die Übersetzung der Konzepte, die im prozeduralen Modell nicht bestehen, ist auch ein wichtiger Schritt zur Überbrückung zweier verschiedener Paradigmen.

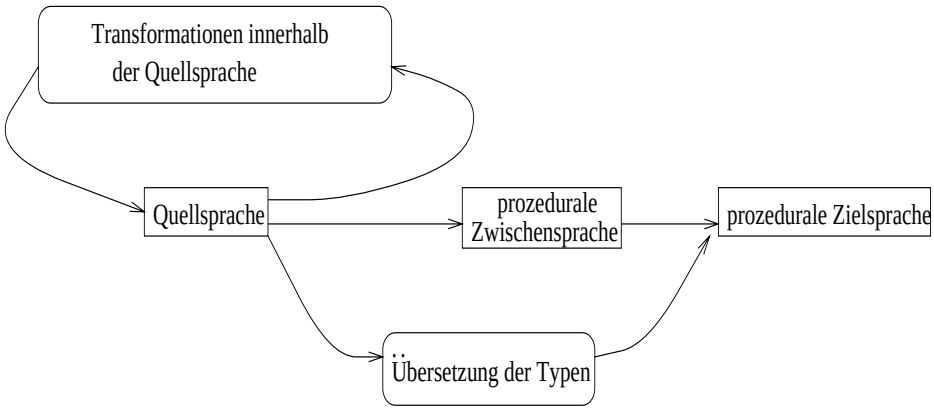
3.2 Quell-, Zwischen- und Zielsprache

Die Quellsprache des Compilers ist eine erweiterte Menge der Lambda-Ausdrücke und die Zielsprache eine prozedurale Sprache. Der Übersetzungsprozeß basiert auf einer zweistufigen Übersetzung: Die Quellsprache wird (nach dem Lambda-Lifting und den *source-to-source* Transformationen innerhalb dieser Sprache) in eine prozedurale Zwischensprache übersetzt, um die Transformation unabhängig von konkreten prozeduralen Sprachen beschreiben zu können.

Die Zwischensprache kann dann abschließend in eine konkrete prozedurale Sprache (bei uns ist das im Beispiel die Sprache *Modula-2*) übersetzt werden. Die Zwischensprache ist keine typisierte Programmiersprache. Der Typübersetzer übersetzt die Typen der Quellsprache in die Typen der prozeduralen Zielsprache (aufgrund der Definition der Übersetzungsfunktion). In der Abbildung 3.1 ist der Prozeß der Übersetzung der Typen und funktionalen Ausdrücke schematisch dargestellt.

Wir trennen die Übersetzung der Quellsprache in eine typfreie prozedurale Zwischensprache von der Übersetzung der Typen, um eine übersichtlichere Darstellung der Transformationsstrategie zu erreichen. Diese Dekomposition der Transformationsprobleme widerspiegelt sich in einer entsprechenden Zerlegung des Compilers.

Abbildung 3.1: Übersetzung der Ausdrücke und Typen



Im folgenden definieren wir die funktionale Quellsprache und die prozedurale Zwischensprache. Die Definition der Sprache *Modula-2*, die wir im Beispiel als prozedurale Sprache verwenden, ist u.a. in [Wir85] zu finden.

Quellsprache

Als funktionale Quellsprache (*FQ*) nehmen wir eine Erweiterung der Menge der Lambda-Ausdrücke, wie sie in Abbildung 2.2 definiert worden sind. In der Abbildung 3.2 fassen wir die Syntax der erweiterten Lambda-Ausdrücke noch einmal zusammen, wobei wir für Funktionsapplikationen aus Darstellungsgründen eine veränderte klammerfreie Syntax einführen.

Die folgende Schreibweise der Applikation einer n-stelligen Funktion

$$\textbf{app}\ f\ arg_1\ \ldots\ arg_n$$

wird als eine vereinfachte Form der Bezeichnung der einstelligen *curried* Funktionsapplikationen

$$(\textbf{app}\ \ldots\ (\textbf{app}\ (\textbf{app}\ f\ arg_1)\ arg_2)\ \ldots\ arg_n)$$

beziehungsweise des folgenden Ausdrucks des Lambda-Kalküls

$$(\ \ldots\ ((f\ \mathrm{arg}_1)\ \mathrm{arg}_2)\ \ldots\ \mathrm{arg}_n)$$

Abbildung 3.2: Syntax der Quellsprache

<*exp*>::=<*constant*>	Konstanten
| <*variable*>	Variablen
| **app** <*exp*> <*exp*>	Funktionsapplikationen
| λ <variable>.<*exp*>	Lambda-Abstraktion
| **let** <*binds*> **in** <*exp*>	Variablenbindungen
| **letrec** <*binds*> **in** <*exp*>	rekursive Definitionen
| **if** <*exp*> **then** <*exp*> **else** <*exp*>	bedingte Ausdrücke
<*binds*> ::= <*bind*> {; <*bind*> }	
<*bind*> ::= <*variable*> = <*exp*>	

betrachtet.

Die Konstanten und Standardfunktionen der Quellsprache sind in der Tabelle in der Abbildung 3.2 nicht angegeben worden. Seien das die ganzen Zahlen, booleschen Konstanten, arithmetischen und logischen Operationen. Mehr über mögliche Erweiterungen des Lambda-Kalküls bis zu einer Basis einer funktionalen Programmiersprache ist in z. B. [BF82] zu finden.

Prozedurale Zwischensprache

Um den Transformationsprozeß weitestgehend unabhängig von einer konkreten prozeduralen Zielsprache gestalten zu können, wird eine prozedurale Zwischensprache eingeführt. Eine direkte Übersetzung der funktionalen Quellsprache *FQ* nach *Modula-2*, die wir in unserer Beispielimplementation als Zielsprache verwenden, wäre mit unserem Algorithmus möglich. In der Abbildung 3.3 ist die Syntax der prozeduralen Zwischensprache (*QTTZ*) angegeben.

Die Syntax der Zwischensprache *QTTZ* ähnelt der Syntax einer einfachen Pascalähnlichen prozeduralen Sprache ohne Konstanten- und Typdeklarationen. Unsere Zwischensprache umfaßt Prozeduren, Anweisungen und Ausdrücke. Prozeduren kennen nur die Parameterart des Werteparameters, während Referenz-Parameter nicht aufgenommen sind. Bei den Anweisungen sind Zuweisungen, IF-Anweisungen sowie Prozeduraufrufe vorhanden. Verschachtelte Prozeduren (lokale Prozeduren) werden unterstützt. Entspreche die Semantik der Sprachelemente von *QTTZ* der üblichen Semantik *Pascal*-artiger Programmiersprachen.

Abbildung 3.3: Die Zwischensprache *QTTZ*

<*programm*>	::= <*prozedur*>
<*prozedur*>	::= PROC <*name*> ARGS <*list_of_vars*> VARS <*list_of_vars*> <*prozeduren*> BEGIN <*anweisungen*> ENDPROC
<*anweisungen*>	::= [<*anweisung*> { <*anweisung* >}]
<*anweisung*>	::= <*if_anweisung* > | <*zuweisung* > | <*funktionsapplikation* >
<*prozeduren*>	::= [*prozedur* {<*prozedur*>}]
<*if_anweisung*>	::= IF <*bool_ausdruck*> THEN <*anweisungen*> [ELSE <*anweisungen*>] ENDIF
<*zuweisung*>	::= ASS <*name*> <*ausdruck* >
<*funktionsapplikation*>	::= APP <*fun_name* > {ARG <*ausdruck*>} ENDAPP
<*fun_name* >	::= <*name* >
<*ausdruck*>	::= <*funktionsapplikation*> | <*konstante*> | <*name*>
<*bool_ausdruck*>	::= <*ausdruck*>
<*list_of_vars*>	::= [*name* {<*name*>}]

3.3 Eine Quelltexttransformation für funktionale Ausdrücke

Im folgenden wird eine Möglichkeit der Transformation funktionaler Ausdrücke in eine prozedurale Sprache angegeben.

Zunächst geben wir die Grundzüge dieser Transformation an.

3.3.1 Grundzüge und Übersetzungsschritte der Transformationsstrategie

Im Rahmen der vorliegenden Dissertation wurde eine Transformationsstrategie für die Überführung von Elementen einer funktionalen Sprache in Elemente einer prozeduralen Sprache entwickelt. Die Hauptidee der neuentwickelten Transformationsstrategie ist die direkte und transparente Übersetzung einer funktionalen in eine prozedurale Sprache.

Grundzüge der Transformationsstrategie

Die Umsetzung unserer Idee basiert auf folgenden Grundprinzipien:

- Es soll eine natürliche Transformation der Kontrollstrukturen erfolgen.
- Die erzeugten Datenobjekte sollen eine möglichst einfache Struktur besitzen (*unboxed*-Datenobjekte sollen unterstützt werden).
- Jede in der funktionalen Sprache definierte Funktion wird in eine Funktion der prozeduralen Sprache übersetzt, wodurch eine Beziehung zum Originalprogramm erhalten bleibt.
- Die Originalbezeichner für Prozeduren, Parameter und Variablen sollen weitestgehend beibehalten werden.
- Jede Standardfunktion, die ihr entsprechendes prozedurales Äquivalent hat, soll in dieses Äquivalent übersetzt werden, z. B.(+ a b) $\Longrightarrow$ a + b.

Übersetzungsschritte

Die Übersetzung der funktionalen Quellsprache in Code einer prozeduralen Sprache erfolgt durch folgende Schritte:

1. Lambda-Lifting

2. *Source-to-source*-Transformationen innerhalb der Quellsprache

3. Transformation des funktionalen Ausdrucks in eine prozedurale Zwischensprache (*QTTZ*)

4. Transformation der *QTTZ*-Zwischensprache in eine konkrete prozedurale Sprache.

Im folgenden wird jeder Schritt näher beschrieben.

3.3.2 Lambda-Lifting

Der funktionale Programmierungsstil unterstützt die Verwendung unbenannter Funktionen (λ). Ziel des Lambda-Liftings ist die Elimination direkter Anwendungen der Lambda-Abstraktionen. Wir beziehen die bekannten Ansätze [Tho85, Jon87] zum Lambda-Lifting als Teil unserer Transformationsstrategie ein.

Mit einem Lambda-Lifting werden alle Lambda-Abstraktionen auf der umgebenden Programmebene als Funktionen definiert und ihnen ein Name zugeordnet. Im Programm wird auf die Namen der Funktionen verwiesen.

Auf diese Weise wird das Programm in der funktionalen Quellsprache nach Anwendung eines Lambda-Liftings die folgende Form haben:

letrec
$\quad a_1 = e_1$
$\quad a_2 = e_2$
$\ldots$
$\quad a_n = e_n$
in e

Die vorher unbenannten Lambda-Abstraktionen e_i erhalten in der **letrec**-Konstruktion einen Namen a_i zugeordnet.

Dabei enthält jeder Ausdruck e_k sowie der Ausdruck e keine Lambda-Abstraktionen als Unterausdrücke. Erläutern wir nun das Verfahren im Detail.

Beim Lifting der Lambda-Abstraktionen unterscheiden wir zwei Fälle:

1. Lifting der geschlossenen Lambda-Abstraktionen (Lambda-Abstraktionen, die keine freien Variablen enthalten) und

2. Lifting der Lambda-Abstraktionen, die freie Variablen haben.

Im folgenden wird das Lambda-Lifting für die beiden Fällen erläutert.

Falls eine Lambda-Abstraktion keine freien Variablen enthält, kann die genannte Transformation ohne Komplikationen durchgeführt werden:

$$(\ldots\ (\lambda\text{x.e})\ \ldots) \Leftrightarrow\!\!\!\rightarrow \quad \begin{array}{ll} \textbf{letrec} & \\ & \$S\ =\ (\lambda x.e) \\ \textbf{in} & (\ldots\$S\ldots) \end{array}$$

Im ursprünglichen Ausdruck (... (λx.e) ...) wird dabei an der Stelle (λx.e) die generierte Variable $S ersetzt.

Zeigen wir nun, wie die Lambda-Abstraktionen mit freien Variablen behandelt werden.

Gegeben sei folgende Lambda-Abstraktion:

λx. ... (λy. ... x ...) ...

Die freie Variable x in der inneren Lambda-Abstraktion wird durch eine β-Abstraktion (siehe Abschnitt 2.1.2) abstrahiert:

$$\lambda\text{x. ..(}\lambda\text{y. ... x ...) ...} \overset{\beta}{\Leftarrow\!\!\!\Rightarrow} \lambda\text{x. ... (}\textbf{app}\ (\lambda\text{wy. ... w ...)\ x) ...}$$

Ordnen wir nun dem Lambda-Ausdruck einen Namen zu (z. B. $\$S$):

$$\$S = (\lambda\text{wy. ... w ...)}$$

Im Quellprogramm tauschen wir die innere Lambda-Abstraktion mit dem Aufruf der Funktion $\$S$ (die durch diese Lambda-Abstraktion definiert wurde) aus:

λx. ... (**app** $S x) ...

Unser Programm sieht nun folgendermaßen aus:

$$\begin{array}{lll} \textbf{letrec} & & \\ & \$S\ =\ (\lambda wy....w...) & \\ \textbf{in} & \lambda\, x. \quad \ldots \quad (\textbf{app} \quad \$S \quad x) & \ldots \end{array}$$

Präzisieren wir nun formal den Algorithmus der Übersetzung des funktionalen Quellprogramms in Superkombinatoren (etwas modifiziertes Verfahren aus [Jon87]):

1. Eine beliebige Lambda-Abstraktion ohne innere Lambda-Abstraktionen auswählen
2. alle ihre freien Variablen als extra Parameter definieren
3. einer neu eingeführten Variablen die Lambda-Abstraktion zuordnen (in der **letrec**-Definition auf globaler Ebene des Programms)

4. die Lambda-Abstraktion mit ihr zugeordnetem Namen ersetzen.

Die Schritte 1 bis 4 sollen solange wiederholt werden, bis es keine inneren Lambda-Abstraktionen im Programm gibt. Auf diese Weise gelangen wir dann nach Anwendung eines Lambda-Liftings zu folgender Form:

letrec
$\$S_1 = Expr_1$
$\$S_2 = Expr_2$
...
$\$S_n = Expr_n$
in *Expr*

Dabei enthält jede Funktionsdefinition keine inneren Lambda-Abstraktionen. Hiermit nähern wir uns bereits der Form eines prozeduralen Programms.

In dem nach dem Lambda-Lifting transformierten Programm wird es partiell parametrisierte Funktionsaufrufe geben. Die partiellen Aufrufe, die durch β-Expansion des Lambda-Lifting entstanden sind[1], werden in folgenden Schritten weiter behandelt: Durch *source-to-source*-Transformationen innerhalb der Quellsprache werden die Fuktionsaufrufe benannt und in **letrec**-Definitionen lokalisiert, damit die Übersetzungsfunktion (die die Quellsprache in die prozedurale Zwischensprache übersetzt) für jeden partiell parametrisierten Aufruf eine neue Funktion definiert.

3.3.3 *Source-to-source*-Transformationen innerhalb der Quellsprache

Vor der Übersetzung in die prozedurale Zwischensprache sind Transformationsschritte innerhalb der funktionalen Quellsprache hilfreich. Dabei wird eine Form des funktionalen Programms angestrebt, von der aus die nachfolgende Übersetzung erleichtert wird.

In einer funktionalen Programmiersprache kann beispielsweise eine **if**-Struktur als Parameter einer Funktion auftreten, was in den meisten prozeduralen Sprachen nicht möglich ist.[2]

Ziel der *source-to-source*-Transformationen ist es, die Anzahl solcher „unerlaubter" Funktionsapplikationen zu minimieren. Im Ergebnis entsteht ein Quellprogramm der Form

[1] Eine generelle Behandlung der partiellen Funktionsapplikationen auf diese Art ist nicht möglich, da es statisch nicht entscheidbar ist, ob es sich um eine partielle Funktionsapplikation handelt.

[2] In *C* gibt es einige Möglichkeiten einer solchen Parametrisierung. Unser Ziel ist aber eine möglichst allgemeine Transformation zu definieren.

letrec
$a_1 = e_1$
$a_2 = e_2$
...
$a_n = e_n$
in e

Dabei gilt folgendes:

- Jede Funktionsapplikation (**app** $f\ x_1\ ...\ x_n$) in den Ausdrücken e_1 ... e_n, e hat an der Stelle der Funktion f eine Variable (also keine anderen Funktionsapplikationen, **if**-, **let**- und **letrec**-Konstruktionen). Die Parameter $x_1\ ...\ x_n$ dürfen keine **if**-, **let**- und **letrec**-Konstrukte sein.

- Die Ausdrücke e_1, e_2, ..., e_n können mit Parametern unterversorgte Funktionsapplikationen sein, aber sie können nicht solche Funktionsapplikationen als Unterausdrücke enthalten. Der Ausdruck e sowie jeder Unterausdruck von e können keine derartigen Funktionsapplikationen sein.

- Die Ausdrücke e_1, e_2, ..., e_n und e sind keine **let**- bzw. **letrec**-Konstruktionen, und der Ausdruck e darf auch keine **if**-Konstruktion sein. Das gilt auch für alle innerhalb der Ausdrücke e_1, e_2, ..., e_n und e liegende **let**- und **letrec**-Konstruktionen.

- Jede **if**-Konstruktion in den Ausdrücken e_1, e_2, ..., e_n und in dem Ausdruck e kann an der Stelle des Booleschen Ausdrucks keine **let**-, **letrec**- bzw. **if**-Konstruktion haben.

Einige der einbezogenen Transformationen stammen aus [San95] (Regeln 3, 4, 7, 8 9, 10), während andere von uns speziell für unser Transformationssystem entwickelt worden sind (Regeln 1, 2, 6, 5, 11). Santos [JeS94, San95] betrachtet seine Transformationsregeln aus der Sicht einer effektiven Implementation der nicht-strikten Sprache *Haskell*. Wir haben diejenigen Regeln übernommen, die auch für unsere Zwecke (Implementation einer strikten Sprache) nutzbar sind.

An einigen Stellen werden neue Variablen eingeführt. Sie beginnen mit dem Zeichnen „$". Darüber hinaus soll in einigen Transformationen wegen der Änderung des Geltungsbereiches von **let**- und **letrec**-Ausdrücken eine Variablenumbenennung durchgeführt werden.

Geben wir nun die Menge der Transformationen an:

1. **Transformation von Funktionsapplikationen, deren Funktion über eine andere Funktionsapplikation ermittelt wird**
   Falls eine Funktion als Ergebnis eine andere Funktion liefert, kann eine Funktionsapplikation durch Verkettung von Funktionsapplikationen erfolgen (die

gelieferte Funktion kann gleich angewendet werden). Solche Verkettung wird folgendermaßen transformiert:

app (**app** *fun rea_param*$_{1_1}$ *... rea_param*$_{1_k}$) *rea_param*$_1$ *rea_param*$_2$ *... rea_param*$_n$

$$\Leftrightarrow\!\!\rightarrow \left\{ \begin{array}{ll} \textbf{letrec} & \\ & \$F \;=\; \textbf{app}\ \mathit{fun}\ \mathit{rea\_param}_{1_1}\ \ldots\ \mathit{rea\_param}_{1_k} \\ \textbf{in} & \textbf{app}\ \$F\ \mathit{rea\_param}_1\ \mathit{rea\_param}_2\ \ldots\ \mathit{rea\_param}_n \end{array} \right.$$

Falls es sich um eine *nicht* rein-funktionale Programmiersprache handelt, so ist es bei dieser Transformation wichtig, daß ein *call-by-value*-Modell des Typs *links-nach-rechts* zugrunde liegt.

Im *rechts-nach-links*-Modell ist die Transformation bei Einbeziehung von prozeduralen Sprachelementen inkorrekt (die Reihenfolge der Berechnung wird mit der Transformation geändert).

Bei einer *rein* funktionalen strikten Sprache, spielt der Berechnungstyp (links nach rechts oder umgekehrt) *keine* Rolle. Die Transformation ist korrekt in beiden Modellen (das globale Terminierungsverhalten ist für die beiden Modelle gleich).

Die von uns definierte Quellsprache *FQ* ist rein-funktional.

Die Korrektheit der Transformation bzw. Berechungsstrategie haben wir wegen der Erweiterungsmöglichkeiten der Sprache mit nicht-funktionalen Elementen diskutiert.

2. **Transformation von partiell parametrisierten Funktionsapplikationen** (*undersaturated function applications*)
   Die Transformation lautet:

   app *fun rea_param*$_1$ *rea_param*$_2$ *... rea_param*$_k$

$$\Leftrightarrow\!\!\rightarrow \left\{ \begin{array}{ll} \textbf{letrec} & \\ & \$F \;=\; \textbf{app}\ \mathit{fun}\ \mathit{rea\_param}_1\ \mathit{rea\_param}_2\ \ldots\ \mathit{rea\_param}_k \\ \textbf{in} & \$F \end{array} \right.$$

   Die Transformation wird nur dann durchgeführt, wenn die Funktion *fun* mit Parametern unterversorgt (*undersaturated*) ist. Mit dieser Transformation werden derartige Funktionsapplikationen in einem **letrec**-Block geschachtelt. Diejenigen Funktionsaufrufe, die schon innerhalb eines **letrec**-Blocks liegen und als Definitionen einer Variablen auftreten, werden nicht transformiert.

Nach dieser Transformation wissen wir, daß alle *undersaturated* Funktionen innerhalb eines **letrec**-Blocks vorkommen. Das erleichtert die weitere Erzeugung des prozeduralen Zielcodes. Die Transformationsregel behandelt nur partielle Applikationen, die statisch erkannt werden können. Partielle Funktionsapplikationen, die nicht statisch erkannt werden können, sind mit unserem Algorithmus nicht übersetzbar.

3. **Das Herausziehen von *let* und *letrec* aus einer Funktionsapplikation**
   Mit dieser Transformation wird eine Funktionsapplikation, deren Funktion durch einen **let**- bzw. **letrec**-Block definiert ist, transformiert. Für **letrec**-Konstrukte sieht die Transformation folgendermaßen aus:

$$\textbf{app}\ (\textbf{letrec}\ \mathit{binds}\ \text{in E})\ \text{args} \quad \Leftrightarrow\!\!\rightarrow \quad \begin{cases} \textbf{letrec} & \\ \quad\quad \mathit{binds} & \\ \quad\quad \$F = E & \\ \textbf{in} \quad \textbf{app}\ \$F\mathit{args} & \end{cases}$$

   Falls in *binds* eine in den Argumenten (*args*) auftretende Variable als linke Seite einer Definition vorkommt, soll eine entsprechende Umbenennung der Variablen durchgeführt werden. Auf die gleiche Weise werden auch die Funktionsapplikationen, deren Funktion durch einen **let**-Block definiert ist, behandelt.

   Die Transformation ist in [San95] (Abschnitt 3.4.1) ausführlich analysiert.

   Falls in einer Transformation **let**- und **letrec**-Blöcke auf die gleiche Art und Weise behandelt werden, bezeichnen wir im folgenden die auf der linken Seite aufgetretenen und auf die gleiche Art und Weise zu behandelnden **let**- und **letrec**-Blöcke mit

   let(rec).

4. **Das Herausziehen von *if*-Konstruktionen aus Funktionsapplikationen**
   Die Transformation ist ähnlich der Transformation der mit **let**- und **letrec**-Blöcken definierten Funktionsapplikationen:

   app (**if** *a* **then** *b* **else** *c*) *args*
   $\Leftrightarrow\!\!\rightarrow$ **if** *a* **then** (**app** *b args*) **else** (**app** *c args*)

   Die Transformation ist in [San95] in Abschnitt 3.5.1 beschrieben.

5. **Das Herausziehen von *let*- und *letrec*-Blöcken aus Argumentenpositionen von Funktionsapplikationen**
   Mit dieser Transformation werden die Argumente einer Funktionsapplikation, die als **let**- bzw. **letrec**- Blöcke definiert wurden, herausgezogen:

$$\textbf{app}\ \mathit{fun}\ \mathit{rea\_param}_1\ \mathit{rea\_param}_2\ \ldots \left(\begin{array}{l}\textbf{let(rec)}\\ \quad \mathit{binds}\\ \textbf{in}\quad \mathit{exp}\end{array}\right) \ldots\ \mathit{rea\_param}_n$$

$$\Leftrightarrow\!\!\rightarrow \left\{\begin{array}{l}\textbf{letrec}\\ \quad\quad \mathit{binds}\\ \textbf{in app}\ \mathit{fun}\ \mathit{rea\_param}_1\ \mathit{rea\_param}_2\ \ldots\ \mathit{exp}\ \ldots\ \mathit{rea\_param}_n\end{array}\right.$$

   Bem.: Der Einflußbereich der Bindungen *binds*, ursprünglich auf *exp* beschränkt, erweitert sich nach der Transformation auf Ausdrücke $\mathit{rea\_param}_1 \ldots \mathit{rea\_param}_n$. Falls eine Variable auf der linken Seite einer Definition in *binds* auftritt, die auch in einem der Ausdrücke $\mathit{rea\_param}_1 \ldots \mathit{rea\_param}_n$ vorkommt, muß sie in *binds* und *exp* einen neuen Namen bekommen.

6. **Das Herausziehen von *if*-Konstrukten aus Argumentenpositionen von Funktionsapplikationen**
   Eine mit einer **if**-Konstruktion parametrisierte Funktionsapplikation wird folgendermaßen transformiert:

$$\begin{array}{l}\textbf{app}\ \mathit{fun}\ \mathit{rea\_param}_1\ \mathit{rea\_param}_2\ \ldots\ (\textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c)\ \ldots\ \mathit{rea\_param}_n\\ \quad \Leftrightarrow\!\!\rightarrow\ \textbf{if}\ a\ \textbf{then}\ (\textbf{app}\ \mathit{fun}\ \mathit{rea\_param}_1\ \mathit{rea\_param}_2\ \ldots\ b\ \ldots\ \mathit{rea\_param}_n)\\ \quad\quad \textbf{else}\ (\textbf{app}\ \mathit{fun}\ \mathit{rea\_param}_1\ \mathit{rea\_param}_2\ \ldots\ c\ \ldots\ \mathit{rea\_param}_n)\end{array}$$

   Die Transformation vergrößert den funktionalen Code. In unserem Modell ist sie wegen der Annäherung an die prozedurale Form des Programms notwendig.

7. **Das Herausziehen von *let* und *letrec* aus dem Definitionsbereich einer anderen *let*- bzw. *letrec*-Konstruktion**
   Bei der Transformation werden die Variablengeltungsbereiche von zwei **let(rec)**-Konstruktionen zusammengesetzt (solche Transformation in einer nicht strikten funktionalen Sprache wurde in [San95], Abschnitt 3.4.2 analysiert):

$$\begin{array}{l}\textbf{let(rec)}\\ \quad\quad binds_v\\ \quad\quad a = \textbf{let(rec)}\ binds_1\ \textbf{in}\ E_1\\ \quad\quad binds_n\\ \textbf{in}\ E\end{array} \quad \Leftrightarrow\!\!\rightarrow \quad \left\{\begin{array}{ll}\textbf{letrec} & \\ & binds_v\\ & binds_1\\ & a\ = E_1\\ & binds_n\\ \textbf{in} & E\end{array}\right.$$

Bem.: Der Einflußbereich der Bindungen $binds_1$, ursprünglich auf E_1 beschränkt, erweitert sich nach der Transformation auf $binds_n$ und E. Im Konfliktfall müssen deshalb Variablen umbenannt werden, d. h. falls eine Variable auf der linken Seite einer Definition in $binds_1$ auftritt, die auch in $binds_n$ oder E vorkommt, muß sie in $binds_1$ und E einen neuen Namen bekommen.

8. **Das Herausziehen von *let* und *letrec* aus dem Ausdruck einer anderen *let*- bzw. *letrec*-Konstruktion**
   Diese Transformation wird in der Theorie der Implementation funktionaler Sprachen als eine Transformation, die den nicht-strikten Sprachen „nichts bringt", bewertet, wenn es sich um die Effizienz des Programms handelt (siehe [San95], Abschnitt 3.4.4). Bei unserem Translator wird die Transformation u.a. wegen der Abwesenheit derartiger Verschachtelung der Variablendefinitionen durchgeführt:

$$\begin{array}{l}\textbf{let(rec)}\\ \quad\quad binds\\ \textbf{in}\quad \textbf{let(rec)}\ binds_1\ \textbf{in}\ E\end{array} \quad \Leftrightarrow\!\!\rightarrow \quad \left\{\begin{array}{ll}\textbf{letrec} & \\ & binds\\ & binds_1\\ \textbf{in} & E\end{array}\right.$$

   Kommen in *binds* und $binds_1$ gleiche Variablen vor, soll eine entsprechende Variablenumbenennung durchgeführt werden.

9. **Das Herausziehen von *if* aus dem Ausdruck von *let* und *letrec***

$$\begin{array}{l}\textbf{let(rec)}\\ \quad binds\\ \textbf{in}\quad \textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c\end{array} \quad \Leftrightarrow\!\!\rightarrow \quad \left\{\begin{array}{ll}\textbf{letrec} & \\ & binds\\ & \$F\ =\ \textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c\\ \textbf{in} & \$F\end{array}\right.$$

   Diese Transformation ist spezifisch für unseren Translator.

10. **Das Herausziehen von *let* und *letrec* aus *if***
    In [San95] ist diese Transformation in Abschnitt 3.4.3. beschrieben. Anstatt den **if**-Ausdruck als Ergebnis von **let** bzw. **letrec** direkt zu liefern, erfolgt bei uns eine Benennung dieses Ausdrucks:

$$\textbf{if} \left(\begin{array}{l} \textbf{let(rec)} \\ \quad \mathit{binds} \\ \textbf{in}\ E \end{array} \right) \textbf{then}\ b\ \textbf{else}\ c$$

$$\Leftrightarrow\!\!\rightarrow \quad \left\{ \begin{array}{ll} \textbf{letrec} & \\ & \mathit{binds} \\ & \$F\ =\ \textbf{if}\ E\ \textbf{then}\ b\ \textbf{else}\ c \\ \textbf{in} & \$F \end{array} \right.$$

11. **Das Herausziehen einer *if*-Konstruktion aus einer anderen *if*-Konstruktion**
Falls der Boolesche Ausdruck einer **if**-Konstruktion eine andere **if** -Konstruktion ist, wird folgende Transformation durchgeführt:

$$\begin{array}{l} \textbf{if}\ \ (\textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c) \\ \textbf{then}\ d \\ \textbf{else}\ e \end{array} \quad \Leftrightarrow\!\!\rightarrow \left\{ \begin{array}{l} \textbf{letrec} \\ \quad \$F = \textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c \\ \textbf{in if}\ \$F\ \textbf{then}\ d\ \textbf{else}\ e \end{array} \right.$$

Hiermit haben wir alle *source-to-source*-Transformationen unserer Quellsprache angegeben. Definieren wir nun, wie die Transformationen angewendet werden.

Jede Transformation hat ihre Priorität. Die Priorität der Transformationen wird mit ihrer Reihenfolge definiert. Die größte Priorität hat die Transformation 1 und die kleinste die Transformation 11. Der Algorithmus zur Anwendung der Transformationen lautet:

1. Wähle Unterausdrücke aus, die einer linken Seite einer Transformationsregel entsprechen (übliches *Top-down-left-to-right-Pattern Matching*).
2. Wende diejenige Transformationsregel auf einen der ausgewählten Unterausdrücke an, die die höchste Priorität hat. Falls mehrere Transformationsregeln mit der höchsten Priorität existieren, so wird der am weitesten links stehende Unterausdruck zuerst ersetzt.

Die Schritte 1 und 2 sollen durchgeführt werden, solange es im Quellprogramm solche Unterausdrücke gibt, die der Form der linken Seite einer beliebigen Transformationsregel entsprechen.

3.3.4 Übersetzung des funktionalen Quellprogramms in die prozedurale Zwischensprache

In diesem Abschnitt werden die Transformationsregeln für die weitere Übersetzung des funktionalen Quellprogramms in die prozedurale Zwischensprache *QTTZ* ange-

geben. Wir haben eine Funktion

$$Trans : FQ \Leftrightarrow\!\!\!\rightarrow QTTZ$$

zu definieren, die alle Ausdrücke der eingeschränkten funktionalen Sprache FQ (nach Lambda-Lifting und *source-to-source*-Transformationen innerhalb der Quellsprache) in prozedurale Programme aus $QTTZ$ überführt.

Dabei definieren wir *Trans* mit Hilfe einer 7-stelligen rekursiven Funktion T, mit der schrittweise elementare Transformationsschritte vorgenomen werden:

$$\mathrm{T}(fun,\ FP,\ \mathrm{E},\ T_e,\ H_e,\ H_v,\ H_p)$$

Die Argumente von T haben folgende Bedeutung:

- fun - Name der Funktion, die durch die Funktion T erzeugt wird
- FP - Liste der formalen Parameter der zu erzeugenden Funktion
- E - Liste, wobei Listenelemente zu übersetzende funktionale Ausdrücke und Schlüsselworte der Sprache $QTTZ$ darstellen
- T_e - in diesem Parameter wird der Programmteil des resultierenden $QTTZ$-Programms schrittweise akkumuliert (das ist eine Liste)
- H_e - Liste, in der die für die kommenden Wertzuweisungen notwendigen Variablen (linke Seiten der kommenden Wertzuweisungen) akkumuliert und nach der Wertzuweisung wieder gestrichen werden
- H_v - akkumuliert alle im Programm (als Liste) vorkommenden Variablen
- H_p - Liste, in der alle Prozeduren (Funktionen) der Zielsprache akkumuliert werden

Die globale Transformationsfunktion *Trans* wird dabei mit Hilfe von T wie folgt definiert:

$$Trans(e) = T(Prog,\ [],\ e,\ [],\ [],\ [],\ [])$$

wobei e ein (eingeschränkter) funktionaler Ausdruck aus *FQ* und *Prog* ein fester reservierter Name für das Hauptprogramm ist.

Bei der Definition der Übersetzungsfunktion werden folgende Listenoperationen verwendet:

- „:" - Listenkonstruktor: a:L erzeugt eine Liste mit dem ersten Element a und der Restliste L. Mit $[a_1, a_2, \ldots, a_n]$ bezeichnen wir die Liste $a_1{:}a_2{:} \ldots a_n{:}[]$. [] ist dabei die Bezeichnung für die leere Liste.
- „\" - $L\backslash a$ liefert eine Liste, die gleich der Liste L ohne das Element a ist (alle Vorkommen von a werden gestrichen). Falls a der Liste L nicht angehört, wird die Liste L geliefert.
- *reverse* - die übliche Funktion zum Umdrehen einer Liste.
- *flatten* - Funktion, deren Argument eine Liste (deren Elemente Listen und Atome sind) ist und als Ergebnis die Liste aller in der Argument-Liste vorkommenden Atome in ungeänderter Reihenfolge liefert (die Funktion annuliert die Listenverschachtelung).

Als nächstes wenden wir uns nun der Definition der Hilfsfunktion T zu.

Übersetzungsregeln:

1. $\mathrm{T}(fun,\ FP,\ \left(\begin{array}{ll} \textbf{letrec} & \\ & a_1 = e_1 \\ & a_2 = e_2 \\ \cdots & \\ & a_n = e_n \\ \textbf{in} & e \end{array}\right)\mathrm{:E},\ T_e,\ H_e,\ H_v,\ H_p)$

   $\Leftrightarrow\!\!\rightarrow\ \mathrm{T}(fun,\ FP,\ \mathrm{ASS{:}e_1{:}\ \ldots\ {:}ASS{:}e_n}{:}e{:}E, T_e,\ \mathrm{a_1{:}\ \ldots\ a_n}{:}H_e,\ \mathrm{a_1{:}\ \ldots\ a_n}{:}H_v,\ H_p)$

   Mit der Transformationregel werden die Variablen $\mathrm{a_1}$ ... $\mathrm{a_n}$ in die Liste H_e aufgenommen und „warten" darauf, daß die Ausdrücke $\mathrm{e_1}$ ... $\mathrm{e_n}$ compiliert werden. Nach der Compilation von $\mathrm{e_1}$ ... $\mathrm{e_n}$ wird der Code der prozeduralen Zwischensprache für die Zuweisung der Werte der compilierten Ausdrücke $\mathrm{e_1}$ ... $\mathrm{e_n}$ an die Variablen $\mathrm{a_1}$ ... $\mathrm{a_n}$ erzeugt. Da die Variablen $\mathrm{a_1}$ ... $\mathrm{a_n}$ in der prozeduralen Zielsprache auch deklariert werden sollen, werden sie ebenfalls in die Liste H_v aufgenommen. Auf die gleiche Art und Weise werden **let**-Konstruktionen behandelt.

2. T(*fun*, *FP*, ASS:($\lambda x_1 ... x_n$.e):E, T_e, a:H_e, H_v, H_p)

   $\Leftrightarrow\!\!\rightarrow$ T(*fun*, *FP*, *E*, T_e, H_e, H_v\a, T(a, [x_1, ..., x_n], e, [], [], [], []):H_p)

   Der Variablen a soll der Wert eines Lambda-Ausdrucks zugewiesen werden. In der prozeduralen Sprache entspricht das einer Funktionsdefinition. Die Variable a wird aus der Liste der Variablen gestrichen und die entsprechende Funktion, deren Name „a" ist, und deren formale Parameter x_1 ... x_n sind, wird definiert.

3. T(*fun*, *FP*, ASS:$\left(\begin{array}{ll} \textbf{letrec} & \\ & a_1 = e_1 \\ & a_2 = e_2 \\ \cdots & \\ & a_n = e_n \\ \textbf{in} & e \end{array}\right)$:E, T_e, a:H_e, H_v, H_p)

   $\Leftrightarrow\!\!\rightarrow$ T(*fun*, *FP*, ASS:e_1:...:ASS:e_n:ASS:*e*:*E*, T_e, a_1:...a_n:a:H_e, a_1:...a_n:H_v, H_p)

   Auf die gleiche Art und Weise übersetzen wir **let**-Konstruktionen im o.g. Kontext.

   Falls einer Variablen a der Wert einer **let**- bzw. **letrec**- Konstruktion zugeordnet werden soll, sollen die Bindungen von **let** bzw. **letrec** durchgeführt werden und daran anschließend die Bindung der Variablen a an den Ausdruck von **let** bzw. **letrec**.

4. T(*fun*, *FP*, ASS:(**if** *cond* **then** *thenopt* **else** *elseopt*):E, T_e, a:H_e, H_v, H_p)

   $\Leftrightarrow\!\!\rightarrow$ T(*fun*, *FP*, IF:*cond*:THEN:ASS:*thenopt*:ELSE:ASS:*elseopt*:ENDIF:*E*, T_e, a:a:H_e, H_v, H_p)

   Falls einer Variablen der Wert eines bedingten Ausdrucks zugewiesen werden soll, wird die entsprechende **if**-Konstruktion compiliert, und der Variablen a werden die **then** bzw. **else**-Kanten zugewiesen.

5. T(*fun*, *FP*, (**if** *cond* **then** *thenopt* **else** *elseopt*):E, T_e, H_e, H_v, H_p)

   $\Leftrightarrow\!\!\rightarrow$ T(*fun*, *FP*, IF:*cond*:THEN:*thenopt*:ELSE:*elseopt*:ENDIF:*E*, T_e, H_e, H_v, H_p)

   Hier wird das funktionale **if** in das prozedurale **IF** übersetzt.

6. T(*fun*, *FP*, ASS:(**app** f arg_1 ... arg_k):E, T_e, a:H_e, H_v, H_p)

   $\Leftrightarrow\!\!\rightarrow$ T(*fun*, *FP*, ASS:arg_1: ... :ASS:arg_k: ASS:(λ x_1 ... x_{n-k}.(**app** f a_{stack_1} ... a_{stack_k} x_1 ... x_{n-k})):E, T_e, a_{stack_1} ... a_{stack_k}:a:H_e, a_{stack_1} ... a_{stack_k}:H_v, H_p)

   falls f eine n-stellige Funktion ist und k<n.

   Die Transformation übersetzt die Zuweisung des Wertes einer partiellen Funktionsapplikation an eine Variable. Die mit k Parametern aufgerufene n-stellige Funktion (n>k) liefert eine (n-k)-stellige Funktion. Die neue Funktion wird durch eine Lambda-Abstraktion definiert. In der prozeduralen Zielsprache wird an dieser Stelle der Code für die Speicherung der Argumente der Funktion erzeugt (die mit *stack* bezeichneten Variablen werden bei der Erzeugung des Zielcodes in einer prozeduralen Sprache besonders behandelt). Die genaue Beschreibung des Verfahrens geben wir nach der Definition der Übersetzungsfunktion.

7. T(*fun*, *FP*, (**app** f arg_1 ... arg_n):E, T_e, H_e, H_v, H_p)

   $\Leftrightarrow\!\!\rightarrow$ T(*fun*, *FP*, APP:f:ARG:arg_1 ... ARG:arg_n:ENDAPP:E, T_e, H_e, H_v, H_p)

   Eine ,,normale" Funktionsapplikation der funktionalen Quellsprache wird in eine Funktionsapplikation der prozeduralen Zielsprache übersetzt.

8. T(*fun*, *FP*, ASS:e:E, T_e, v:H_e, H_v, H_p)

   $\Leftrightarrow\!\!\rightarrow$ T(*fun*, *FP*,e:E, v:ASS:T_e, H_e, H_v, H_p)

   In dieser Situation soll der Code für die Zuordnung des Wertes eines Ausdrucks (e) an eine Variable (v), die der Kopf der Liste H_e ist, erzeugt werden. Die Variable und das Schlüsselwort (ASS) werden in die Liste T_e aufgenommen. Danach wird der Ausdruck e übersetzt, und anschließend wird er auch in die Liste T_e aufgenommen. Damit wird die Zuweisung in der Sprache $QTTZ$ vollendet.

9. T(*fun*, *FP*, *ord*:E, T_e, H_e, H_v, H_p)

   $\Leftrightarrow\!\!\rightarrow$ T(*fun*, *FP*, E, *ord*:T_e, H_e, H_v, H_p)
   Diese Transformation wird angewendet, falls $ord \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{V}$.
   Dabei haben die o.g. Bezeichner folgende Bedeutung:
   $\mathcal{P}$ - Menge der Schlüsselworte der Sprache $QTTZ$
   $\mathcal{C}$ - Menge der Konstanten des von uns definierten erweiterten Lambda-Kalküls
   $\mathcal{V}$ - Menge der Variablen.

Wenn die zu übersetzende Konstruktion eine Konstante, Variable oder ein *QTTZ*-Schlüsselwort ist, wird sie einfach in der Liste T_e abgelegt.

10. T(*fun*, *FP*, [], T_e, H_e, H_v, H_p)

    $\Longleftrightarrow$ *flatten*([PROC, *fun*, ARGS, *FP*, VARS, H_v, H_p, BEGIN, *reverse*(T_e), ENDPROC])

    Der zu übersetzende funktionale Ausdruck ist „leer". Der Name der zu erzeugenden Funktion ist *fun*, die Liste ihrer Parameter ist *FP*, die Liste der Variablen ist H_v. H_p und T_e sind die Listen der lokalen Prozeduren bzw. der Programmteil. Die Teile des *QTTZ*-Programms werden in den entsprechenden Bereichen abgelegt.

In der Regel 6 haben wir die speziellen Variablen mit der Bezeichnung *stack* eingeführt. Bei der Erzeugung des prozeduralen Zielprogramms werden diese Variablen folgendermaßen behandelt (Regeln für die Stackvariablen):

1. ASS:a_{stack}:*expr* bedeutet, daß der Variablen a_{stack} der Wert *expr* zugeordnet und die Variable in den Stack des Laufzeitsystems aufgenommen wird.

2. Nach dem Aufruf der Funktion, die als eine *undersaturated* Funktionsapplikation definiert wurde, soll die Variable a_{stack} vom Stack gestrichen werden (sie lebt nur in der Laufzeit der ihr entsprechenden Funktion).

3. Durch die partiell parametrisierte Funktion wird eine neue Funktion definiert. Im Rumpf der neuen Funktion befindet sich ein Aufruf der Funktion, die partiell parametrisiert wurde. Diese Funktionsapplikation enthält *Stackvariablen*. Vor dem Aufruf sollen alle in der Funktionsapplikation vorkommenden Variablen vom *Stack* genommen werden. Dieses Verfahren ist nur für die statisch erkennbaren partiellen Funktionsapplikationen gültig.

Im Beispiel am Ende dieses Abschnitts ist konkret gezeigt, wie Stackvariablen behandelt werden.

3.3.5 Übersetzung von *QTTZ* in eine konkrete prozedurale Sprache

Zeigen wir nun, wie ein Programm in der Sprache *QTTZ* in eine prozedurale Sprache übersetzt werden kann. Als konkrete prozedurale Sprache wird die Sprache *Modula*-2 genommen. Ein *QTTZ*-Programm hat die folgende Form:

PROC *Name* ARGS $form\_par_1$ ... $form\_par_n$
VARS Var_1 ... Var_m

BEGIN *Code*
ENDPROC

Ein derartiges *QTTZ*-Programm wird folgendermaßen nach *Modula-2* übersetzt:

PROC *Name* ARGS $form\_par_1$ ... $form\_par_n$
VARS Var_1 ... Var_m
BEGIN *Code*
ENDPROC $\Leftrightarrow\rightarrow$

$$\Leftrightarrow\rightarrow \begin{cases} PROCEDURE\ Name(form\_par_1 : Type(form\_par_1);\ ... \\ \qquad\qquad form\_par_n : Type(form\_par_n)) : Type(Name); \\ VAR \\ \qquad Var_1 : Type(Var_1);\ ...\ Var_m :\ Type(Var_m); \\ BEGIN \\ \qquad [stack\_manipulation;] \\ \qquad T_{M2}(Code) \\ ENDName; \end{cases}$$

Die Funktion *Type* liefert für jede Variable die ihr entsprechenden Typen in *Modula-2* und stützt sich auf den Typübersetzer, der im Abschnitt 3.4 definiert wird.

Bei der Übersetzung der partiell parametrisierten Funktionen haben wir die *Stack*-Variablen definiert. Hängt eine Funktion von *Stack*-Variablen ab, stellt *stack_manipulation* den Code für das Lesen der entsprechenden Variablen vom *Stack* (siehe Abschnitt 3.3.4, Regel 6 und Regeln für die Behandlung der Stackvariablen am Ende des Abschnitts). Die Übersetzungsfunktion nennen wir T_{M2}:

$$T_{M2} : QTTZ \rightarrow Modula\text{-}2$$

Der Definitionsbereich der Funktion ist die Sprache *QTTZ* und der Wertebereich Sprachkonstruktionen der Sprache *Modula-2*. Zusätzlich wird noch eine Hilfsfunktion T_{nsM2} definiert, die eine ähnliche Semantik wie die Funktion T_{M2} hat. Sie wird für die korrekte Handhabung der Semikolons genutzt. Zeigen wir nun, wie die Konstrukte von *QTTZ* nach *Modula-2* durch die Funktionen T_{M2} und T_{nsM2} übersetzt werden.

IF, THEN, ELSE, ENDIF

Die Konstrukte haben ihre Äquivalente in *Modula-2* und werden demnach direkt übersetzt:

T_{M2}(IF e_1 THEN e_2 ELSE e_3 ENDIF)$\Leftrightarrow\!\!\rightarrow$
IF $T_{nsM2}(e_1)$ THEN $T_{nsM2}(e_2)$ ELSE $T_{nsM2}(e_3)$ END;

T_{nsM2}(IF e_1 THEN e_2 ELSE e_3 ENDIF)$\Leftrightarrow\!\!\rightarrow$
IF $T_{nsM2}(e_1)$ THEN $T_{nsM2}(e_2)$ ELSE $T_{nsM2}(e_3)$ END

ASS

Die Zuordnung eines Wertes an eine Variable wird folgendermaßen übersetzt:

T_{M2}(ASS x e) $\Leftrightarrow\!\!\rightarrow$ x := T_{M2}(e) [*stack_opt* ;]

T_{nsM2}(ASS x e) $\Leftrightarrow\!\!\rightarrow$ x := T_{nsM2}(e) [;*stack_opt*]

Falls die Variable x eine *Stack*-Variable ist, bedeutet *stack_opt* die Aufnahme der Variablen auf dem *Stack*. Zum Beispiel:
T_{nsM2}(ASS a_{stack} e) $\Leftrightarrow\!\!\rightarrow$ a_{stack} := T_{nsM2}(e) ; Push(*Stack*, a_{stack})

wobei *Stack* der globale Variablenstack des erzeugten Programms ist.

APP, ARG, ENDAPP

Die Übersetzung der Funktionsapplikationen der Sprache *QTTZ* in *Modula-2* lautet:

T_{M2}(APP *fun* ARG arg_1 ARG arg_2 ... ARG arg_n ENDAPP) $\Leftrightarrow\!\!\rightarrow$
$T_{nsM2}(fun)($ $T_{nsM2}(arg_1)$, $T_{nsM2}(arg_2)$, ..., $T_{nsM2}(arg_n)$);

T_{nsM2}(APP *fun* ARG arg_1 ARG arg_2 ... ARG arg_n) $\Leftrightarrow\!\!\rightarrow$
$T_{nsM2}(fun)($ $T_{nsM2}(arg_1)$, $T_{nsM2}(arg_2)$, ..., $T_{nsM2}(arg_n)$)

Bezeichner und Spezialzeichen

Modula-2-Äquivalente der Bezeichner und Spezialzeichen aus *QTTZ* sind die gleichen Bezeichner bzw. Spezialzeichen:

$T_{M2}(bss) \Leftrightarrow\!\!\rightarrow bss$;

$T_{nsM2}(bss) \Leftrightarrow\!\!\rightarrow bss$

Dabei ist *bss* ein Bezeichner bzw. Spezialzeichen.

Zeigen wir nun ein Beispiel der Übersetzung eines funktionalen Programms nach (*QTTZ*) und den daran anschließend erzeugten Code in *Modula-2*.

Beispiel

Sei das zu übersetzende funktionale Programm eine Funktionsapplikation mit einer durch eine **let**-Definition definierten Funktion:

$$\mathbf{app}\left(\begin{array}{ll}\mathbf{let} & plusfun = \lambda\, x\, y.\mathbf{app} + x\, y\\ \mathbf{in} & (\mathbf{app}\, plusfun\, 10)\end{array}\right) 20$$

Da es im Programm keine Lambda-Abstraktionen als Unterausdrücke gibt (sie erscheinen nur als rechte Seiten der Variablendefinitionen von **let**-Konstruktionen), werden die *source-to-source*-Transformationen gleich durchgeführt. Wir bezeichnen die Transformationsschritte mit dem Nummer der entsprechenden *source-to-source*-Transformation.

$$\mathbf{app}\left(\begin{array}{ll}\mathbf{let} & plusfun = \lambda\, x\, y.\mathbf{app} + x\, y\\ \mathbf{in} & (\mathbf{app}\, plusfun\, 10)\end{array}\right) 20 \xrightarrow{2} \mathbf{app}\left(\begin{array}{l}\mathbf{let}\, plusfun = \lambda\, x\, y.\mathbf{app} + x\, y\\ \mathbf{in}\left(\begin{array}{ll}\mathbf{letrec}\\ \quad \$f & = \mathbf{app}\, plusfun\, 10\end{array}\right)\\ \mathbf{in}\, \$f\end{array}\right) 20$$

$$\xrightarrow{3}\left(\begin{array}{ll}\mathbf{letrec} & plusfun = \lambda\, x\, y.\mathbf{app} + x\, y\\ \quad \$F & = \left(\begin{array}{ll}\mathbf{letrec} & \$f = \mathbf{app}\, plusfun\, 10\\ \mathbf{in} & \$f\end{array}\right)\\ \mathbf{in} & \mathbf{app}\, \$F\, 20\end{array}\right) \xrightarrow{7}\left(\begin{array}{ll}\mathbf{letrec} & plusfun = \lambda\, x\, y.\mathbf{app} + x\, y\\ \quad \$f & = \mathbf{app}\, plusfun\, 10\\ \quad \$F & = \$f\\ \mathbf{in} & \mathbf{app}\, \$F\, 20\end{array}\right)$$ [3]

Das mit *source-to-source*-Transformationen ermittelte Programm wird nun mit der definierten Übersetzungsfunktion *Trans* in die prozedurale Zwischensprache übersetzt. Hier beziehen wir die entsprechenden Nummern der Transformationsregeln aus Abschnitt 3.3.4 ein.

$$Trans\left(\begin{array}{ll}\mathbf{letrec} & plusfun = \lambda\, x\, y.\mathbf{app} + x\, y\\ \quad \$f & = \mathbf{app}\, plusfun\, 10\\ \quad \$F & = \$f\\ \mathbf{in}\, \mathbf{app}\, \$F\, 20\end{array}\right) =$$

$$= \mathrm{T}(\mathrm{Prog}, [], \left[\left(\begin{array}{ll}\mathbf{letrec} & plusfun = \lambda\, x\, y.\mathbf{app} + x\, y\\ \quad \$f & = \mathbf{app}\, plusfun\, 10\\ \quad \$F & = \$f\\ \mathbf{in}\, \mathbf{app}\, \$F\, 20\end{array}\right)\right], [], [], [], [])$$

[3] Falls die *source-to-source*-Transformationen in einer anderen Reihenfolge angewendet worden wären, könnte das Zielprogramm einfacher sein:

$$\mathbf{app}\left(\begin{array}{ll}\mathbf{let} & plusfun = \lambda\, x\, y.\mathbf{app} + x\, y\\ \mathbf{in} & (\mathbf{app}\, plusfun\, 10)\end{array}\right) 20 \xrightarrow{3}\left(\begin{array}{ll}\mathbf{letrec} & plusfun = \lambda\, x\, y.\mathbf{app} + x\, y\\ \mathbf{in} & (\mathbf{app}\ (\mathbf{app}\, plusfun\, 10)\ 20)\end{array}\right) \xrightarrow{1,8}$$
$$\left(\begin{array}{ll}\mathbf{letrec} & plusfun = \lambda\, x\, y.\mathbf{app} + x\, y\\ \quad \$F & = \mathbf{app}\, plusfun\, 10\\ \mathbf{in} & \mathbf{app}\, \$F\, 20\end{array}\right).$$

Die Optimierung der Menge der *source-to-source*-Transformationen wird hier nicht analysiert.

$\stackrel{1}{\rightarrow}$ T(Prog, [] , [ASS,(λ x y . **app** + x y),ASS,(**app** plusfun 10),ASS,$f,(**app** $F 20)], [], [plusfun,$f,$F], [plusfun,$f,$F], [])

$\stackrel{2}{\rightarrow}$ T(Prog, [] , [ASS,(**app** plusfun 10),ASS,$f,(**app** $F 20)], [], [$f,$F], [$f,$F], [$T_1$])

wobei T_1 = T(plusfun, [x,y], [(**app** + x y)], [], [], [], [])

$\stackrel{6}{\rightarrow}$ T(Prog, [] , [ASS,10,ASS, (λ x. **app** plusfun a_{stack} x),ASS,$f,(**app** $F 20)], [], [$a_{stack}$,$f,$F], [$a_{stack}$,$f,$F], [$T_1$])

$\stackrel{8,9}{\rightarrow}$ T(Prog, [] , [ASS,(λ x. **app** plusfun a_{stack} x),ASS,$f,(**app** $F 20)], [10,$a_{stack}$,ASS], [$f,$F], [$a_{stack}$,$f,$F], [$T_1$])

$\stackrel{2}{\rightarrow}$ T(Prog, [], [ASS,$f,(**app** $F 20)], [10,$a_{stack}$,ASS], [$F], [a_{stack},$F], [$T_2, T_1$])

wobei T_2 =T($f, x, [**app** plusfun a_{stack} x], [], [], [], [])

$\stackrel{8,9}{\rightarrow}$ T(Prog, [], [**app** $F 20], [$f,$F,ASS,10,$a_{stack}$,ASS], [], [$a_{stack}$,$F], [T_2, T_1])

$\stackrel{7}{\rightarrow}$ T(Prog, [], [APP,$F,ARG,20,ENDAPP], [$f,$F,ASS,10,$a_{stack}$,ASS], [], [$a_{stack}$,$F], [T_2, T_1])

$\stackrel{9(5x)}{\rightarrow}$ T(Prog, [], [], [ENDAPP,20,ARG,$F,APP,$f,$F,ASS,10,$a_{stack}$,ASS], [], [$a_{stack}$,$F], [T_2, T_1])

$\stackrel{10}{\rightarrow}$*flatten*([PROC,Prog,ARGS,VARS,a_{stack},$F,$T_2$,$T_1$,BEGIN, *reverse*([ENDAPP,20,ARG,$F,APP,$f,$F,ASS,10,$a_{stack}$,ASS]),ENDPROC])

wobei T_1 = ([PROC,plusfun,ARGS,x,y,VARS,BEGIN,APP,+,ARG,x, ARG,y,ENDAPP,ENDPROC])

und T_2 = ([PROC,$f,ARGS,x,VARS,BEGIN,APP,plusfun,ARG, a_{stack},ARG,x,ENDAPP,ENDPROC])

$\rightarrow$ [PROC,Prog,ARGS,VARS,a_{stack},$F, PROC,$f,ARGS,x,VARS,BEGIN,APP,plusfun,ARG, a_{stack},ARG,x,ENDAPP,ENDPROC,PROC,plusfun,ARGS,x,y,VARS,BEGIN, APP,+,ARG,x,ARG,y,ENDAPP,ENDPROC,BEGIN,ASS,a_{stack},10,ASS,$F,$f, APP,$F,ARG,20,ENDAPP,ENDPROC]

Die abschließende Übersetzung in *Modula-2* (vereinfacht) ergibt:

```
MODULE beispiel1;
FROM InOut IMPORT
  WriteString, WriteLn, WriteInt;
FROM STACK IMPORT
  Push, Pop, Top, MakeNull, Stack;

TYPE
  proctype = PROCEDURE(INTEGER):INTEGER;

VAR
  Astack, Prog : INTEGER;
  dollarF      : proctype;
  S            : Stack;

PROCEDURE dollarf(x:INTEGER):INTEGER;
BEGIN
  Top(S, Astack); (* Stackvariable-Regel 3 *)
  RETURN plusfun(Astack, x)
END dollarf;

PROCEDURE plusfun(x, y: INTEGER):INTEGER;
BEGIN
  RETURN x+y
END plusfun;

BEGIN
  MakeNull(S);
  Astack := 10;
  Push(S, Astack); (* Stackvariable-Regel 1 *)
  dollarF := dollarf;
  Prog  := dollarF(20) ;
  Pop(S); (* Stackvariable-Regel 2 *)
  WriteInt(Prog, 4);
END beispiel1.
```

In diesem Abschnitt haben wir einen Transformationsalgorithmus für die Übersetzung einer strikten funktionalen Sprache in eine prozedurale Sprache definiert. Im Anhang B ist die Transformation mehrerer Beispielprogramme angegeben.

Im folgenden Abschnitt wird eine Möglichkeit der Übersetzung funktionaler Programme mit einer nicht-strikten Auswertung analysiert.

3.3.6 Erzwungene und verzögerte Evaluation

In diesem Abschnitt werden die grundlegenden Begriffe der Auswertungsstrategie erläutert und eine Möglichkeit der Realisierung einer verzögerten Verarbeitung unendlicher Listen in *Modula-2* vorgestellt. Auf ähnliche Art und Weise wäre es möglich, über die Verarbeitung unendlicher Listen hinausgehende Modelle der *lazy evaluation* im prozeduralen Modell zu realisieren. Für eine generelle Realisierung der Evaluation im prozeduralen Modell wird die im Kapitel 5 dargestellte Technik empfohlen.

Das in diesem Abschnitt angegebene Verfahren ist aus der Literatur bekannt [BN89, GP84, MB97] und stellt häufig einen Teil des Laufzeitsystems eines Compilers einer funktionalen Programmiersprache dar. Wir nutzen das Verfahren in einem anderen Zusammenhang, in dem wir es an die Bedingungen der direkten Übersetzung in eine prozedurale Sprache anpassen.

Strikte und nicht-strikte Funktionen

Definieren wir zunächst die Eigenschaft der Striktheit von Funktionen. Dabei verwenden wir die spezielle Notation $\perp$ für einen Ausdruck, dessen Evaluation nicht terminiert.

Definition. *Eine einstellige Funktion ist strikt genau dann, wenn:*

$$f \perp = \perp.$$

Es ist nun leicht, die Definition auf Funktionen mit mehreren Argumenten auszudehnen. Sei g eine Funktion mit n Argumenten (n>1). Die Funktion g ist strikt in ihrem k-ten Argument genau dann, wenn:

$$g\ arg_1\ arg_2\ ..\ arg_{k-1}\ \perp\ arg_{k+1}\ ..\ arg_a = \perp.$$

Wenn eine Funktion nicht-strikt ist, so wird sie als *lazy* bezeichnet.[4]

Auswertungsstrategien

Bei funktionalen Sprachen gibt es zwei grundlegende Strategien der Auswertung:

1. **Strikte Auswertung** *(leftmost-innermost, call-by-value, application order reduction, eager evaluation)*:
   Die Argumente einer aufgerufenen Funktion werden ausgewertet, bevor die Abbildungsvorschrift der Funktion angewendet wird .

2. **Nicht-strikte Auswertung** *(leftmost-outermost, call-by-name, normal order reduction)*:
   Die Abbildungsvorschrift der Funktion wird ausgewertet, bevor die Argumente der Funktion ausgewertet werden.

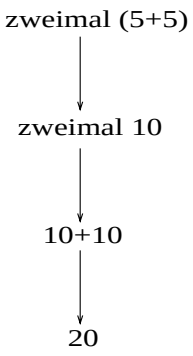
Geben wir als Beispiel die Funktion *zweimal* an, die folgendermaßen definiert ist[5]:

zweimal :: Int->Int
zweimal x = x+x

Im strikten Auswertungsmodell wird der Funktionsaufruf *zweimal* (5+5) folgendermaßen berechnet:

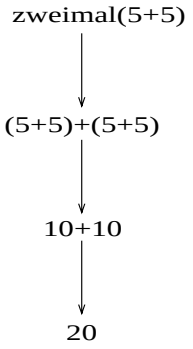
[4]Diese Terminologie ist nicht völlig korrekt, weil *lazy evaluation* eine Auswertungsstrategie für die Implementation der nicht-strikten Semantik einer Programmiersprache ist.

[5]Hier wird die Haskell-Syntax benutzt. Die Definition der Funktion *zweimal* ist einfach und wird nicht weiter erläutert.



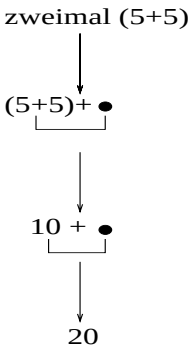
Zunächst werden die Argumente der Funktion *zweimal* berechnet. Danach wird die Funktion selbst angewendet.

Im nicht-strikten Modell läuft die Berechnung folgendermaßen ab:



In einem nicht-strikten funktionalen Modell wird nur das berechnet, was ,,notwendig ist". In diesem Beispiel kommt ein Nachteil dieser Strategie zu Vorschein: Ausdrücke müssen u. U. mehrmals (z. B. (5+5)) berechnet werden.

In der folgenden Abbildung ist eine Optimierung gezeigt [Hud89]:



Diese optimierte Auswertungsstrategie heißt *call-by-need* oder *lazy evaluation*. Die *call-by-name* und *lazy evaluation* Auswertungen sind besonders nützlich bei der Erzeugung unendlicher Datenstrukturen (Listen).

In der Abbildung 3.4 ist eine funktionale Definition der Liste der natürlichen Zahlen und der Aufruf einer Funktion, die das zehnte Element der Liste zurückgibt, angegeben. Dieses Programm verlangt eine nicht-strikte Verarbeitung. Die Listen, die eine nicht-strikte Auswertung fordern, nennen wir nicht-strikte Listen.[6]

Abbildung 3.4: Beispiel der Definition einer unendlichen Liste

```
nat :: Int -> [Int]
nat n = n : nat (n+1)


nthel :: Int -> [a] -> a
nthel n (x:xs) = if n==1 then x else nthel (n-1) xs

main = nthel 10 (nat 1)
```

Die unendlichen Listen verdienen bei den funktionalen Programmiersprachen besondere Aufmerksamkeit, weil viele funktionale Konzepte wie z. B. rein-funktionale Ein- und Ausgabe oftmals durch diese Listen bzw. Ströme (engl. *streams*) realisiert werden.

Im folgenden wird die Möglichkeit der Implementation der nicht-strikten Listen im prozeduralen Programmodell analysiert.

Unendliche Listen im prozeduralen Modell

Die in der Abbildung 3.4 angegebene Funktion *nat* ist im prozeduralen Modell wegen der dortigen strikten Auswertung auf diese Art und Weise nicht definierbar. Die Implementation unendlicher Listen im prozeduralen Programmodell muß auf anderen Grundprinzipien aufbauen:

- Die Konstruktion einer unendlichen („trägen") Liste muß unterbrochen werden, obwohl die Liste, die im Moment der Auswertung konstruiert wird, noch nicht vollständig ist (die für die Liste wichtigen Angaben müssen gespeichert werden).

[6] Genauer gesagt, handelt es sich um unendliche Listen.

- Bei der Bearbeitung der generierten „trägen" Liste muß in Betracht gezogen werden, daß die Liste nicht ausgewertet wurde. Vor den entsprechenden Listenoperationen (z. B. head, tail) müssen benötigte Elemente der Liste ausgerechnet werden.

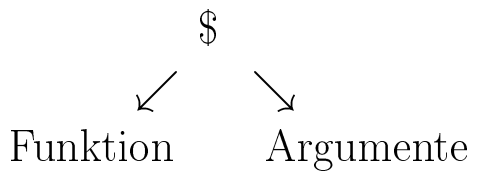
Definieren wir nun Funktionen, Prozeduren und Datenstrukturen für eine solche Behandlung von Listen. Diese Komponenten gehören zu den grundlegenden Teilen des Laufzeitsystems eines Compilers einer funktionalen Sprache.

Laufzeitsystem für die Listenbearbeitung

Bei unserer Implementation unendlicher Listen im prozeduralen Programmodell besteht die Hauptidee darin, die Listenberechnung zu unterbrechen und den Berechnungszustand abzuspeichern. In der nächsten Situation, in der die nicht fertig berechnete Liste notwendig ist, wird das nächste Element der Liste mit Hilfe der gespeicherten Funktion und der Argumente ausgerechnet.

In der Abbildung 3.5 ist eine für die unendlichen Listen geeignete Struktur dargestellt. Ihren Konstruktor $ nennen wir *Lazy Node*.

Abbildung 3.5: Die Hauptstruktur der unendlichen Liste



Funktion - Funktion, mit der der Rest der Liste ausgewertet wird
Argumente - Argumente der Funktion
$ - Konstruktor der Struktur

Für die dargestellte Datenstruktur der Listenrepräsentation müssen wir noch die Menge der Funktionen für die Listenbearbeitung definieren. Die notwendigen Funktionen sind:

- PROCEDURE HeadSE(L: SExp): SExp;
  Funktion, die der Funktion Head entspricht,

- PROCEDURE TailSE(L: SExp): SExp;
  Pendant für die Funktion Tail,

- PROCEDURE ConsSE(Car: SExp; Cdr: SExp): SExp;
  Funktion für die übliche Funktion Cons einer funktionaler Sprache,

- PROCEDURE NewLazySE(Fun: ADDRESS; Arg: SExp): SExp;
  Funktion für die Erzeugung von *Lazy Node*,

- PROCEDURE EvalLazy(VAR LazyList: SExp);
  Prozedur für die Auswertung von *Lazy Node*,

- PROCEDURE LazyHeadSE(L: SExp): SExp;
  Funktion, die der Funktion Head entspricht und das Verfahren für die Berechnung von *Lazy Node* enthält,

- PROCEDURE LazyTailSE(L: SExp): SExp;
  „Lazy"-Pendant für die Funktion Tail,

- PROCEDURE LazyConsSE(Car: SExp; VAR CdrFun: ARRAY OF BYTE; VAR CdrArgs: ADDRESS): SExp;
  Entsprechende Funktion für die übliche Funktion Cons,

SExp ist dabei der Typ, der der in *Modula-2* implementierten Datenstruktur der Listen entspricht. Bei der Erzeugung von *Lazy Node* müssen die Funktion und ihre Argumente bekannt sein. Die Funktion NewLazy gibt eine Struktur zurück, die der Struktur in der Abbildung 3.5 entspricht.

Die Auswertung von *Lazy Node* bedeutet die Anwendung der Funktion auf die Argumente, die in der Struktur gespeichert wurden und die Erzeugung eines neuen Knotens - des Ergebnisses der Auswertung (**EvalLazy**).

Bei der Bearbeitung einer Liste (mit Head und Tail) ist es notwendig, zunächst zu überprüfen, ob die Liste ein *Lazy Node* ist, und den Knoten vor dem Zurückgeben des Ergebnisses auszuwerten (**LazyHeadSE**, **LazyTailSE**).

Im ganzen Mechanismus der Listenbearbeitung spielt die Konstruktion der Listen eine wichtige Rolle. Bei der Technik der Implementation funktionaler Programmiersprachen z. B. durch SK-Graph-Reduktion wird an dieser Stelle die Berechnung der abstrakten Maschine unterbrochen und der nächste zu reduzierende Knoten des Graphen reduziert. Bei unserer Implementation wurde dieses Verhalten durch die Konstruktion der Liste mit Funktionen und Argumenten ohne Auswertung (die zur unendlichen Abarbeitung des Programms führt) realisiert (**LazyConsSE**).

Eine Möglichkeit der Übersetzung von force und delay - Konstruktionen der Sprache *Scheme* in Modula-2

Die Bearbeitung unendlicher Strukturen wird in *Scheme* mit Hilfe der Spezialformen **force** und **delay** ermöglicht. Eine Übersetzung dieser Strukturen von *Scheme* nach Modula-2 kann durch *Lazy Nodes* realisiert werden. Wir verwenden den folgenden funktionalen *Scheme*-Ausdruck (Scheme-Programm) als Beispiel:

```
(letrec
  ((odds
     (lambda (n)
       (cons n (delay (odds (+ n 2))))))

   (nthlazy
     (lambda (n l)
        (if (eqv? n 1)
           (car l)
           (nthlazy (- n 1) (force (cdr l)))))))
  (nthlazy 5 (odds 1)))
```

In diesem Programm wird das fünfte Element einer unendlichen Liste (Liste der ungeraden Zahlen) berechnet. Mit **delay** wird die Berechnung der Liste unterbrochen, während mit **force** die Berechnung der unendlichen Liste erzwungen wird (da ein unausgewertetes Element der Liste notwendig ist). **delay** und **force** haben explizit ihre Äquivalente im schon beschriebenen System. Es handelt sich dabei um Funktionen **EvalLazyFun** und **NewLazySE**:

```
PROCEDURE EvalLazyFun(LazyList: SExp): SExp;
BEGIN
  EvalLazy(LazyList);
  IF IsTypeSE(LazyList, ConstrSE) THEN
    EvalLazy(HeadSE(LazyList));
    EvalLazy(TailSE(LazyList))
  END;
  RETURN LazyList
END EvalLazyFun;
```

Diese Funktion ist ein funktionales Äquivalent der schon definierten Funktion EvalLazy. Die zweite Funktion ist die schon beschriebene Funktion **NewLazySE**:

```
PROCEDURE NewLazySE(Fun: ADDRESS;  Arg: SExp ): SExp;
VAR
  Hlp: SExp;
BEGIN
  NewSE(Hlp);
```

```
    SetTypeSE(Hlp, LazySE);
    ALLOCATE(Hlp^.LazyEl, TSIZE(LazyNode));
    Hlp^.LazyEl^.Function := Fun;
    Hlp^.LazyEl^.Arguments := Arg;
    RETURN Hlp;
  END NewLazySE;
```

Die Funktion **EvalLazyFun** erzwingt die Berechnung des unausgewerteten Knotens und gibt als Resultat den evaluierten Knoten zurück. **NewLazySE** unterbricht den nicht-strikten Ablauf des Programms ähnlich wie **delay**. Wir geben abschließend das Ergebnis der Übersetzung des oben angegebenen *Scheme*-Programms nach der vorgeschlagenen Strategie an:

```
MODULE nth5
FROM SExps IMPORT
   DispSE, ConSE, HeadSE,TailSE, ConsSE, QIntSE, ValIntSE, InitSE;
FROM SExpsKr3 IMPORT
   SExp, FunTypePtr, FunType  ;

FROM Lazy IMPORT
  LazyConsSE, LazyHeadSE, LazyTailSE, NewLazySE, EvalLazyFun;

FROM SYSTEM IMPORT
  ADR, TSIZE;

FROM Storage IMPORT
  ALLOCATE, DEALLOCATE;

VAR
  OddFunPtr: FunTypePtr;
  OddFun: FunType;


PROCEDURE ODDS (N:SExp):SExp;
BEGIN
  RETURN ConsSE(N, NewLazySE(OddFunPtr,
                             QIntSE((ValIntSE(N)+2))))
END ODDS ;

PROCEDURE NTHLAZY (N: INTEGER ;
               L: SExp):SExp ;
BEGIN
  IF (N=1) THEN
    RETURN HeadSE (L)
  ELSE
    RETURN NTHLAZY((N-1), EvalLazyFun(TailSE(L)))
  END ;
END NTHLAZY ;
BEGIN
  InitSE ;
  OddFun := ODDS;                       (* ! *)
  OddFunPtr := FunTypePtr(ADR(OddFun)); (* !! *)
  DispSE(NTHLAZY(5, ODDS(QIntSE(1))))
END nth5.
```

Der Code in den Zeilen (* ! *) und (* !! *) des Hauptprogramms stellt die Vorbereitung des Systems auf die Speicherzuordnung von *Lazy Node* dar.

In diesem Abschnitt haben wir nur die Möglichkeit der verzögerten Auswertung im prozeduralen Modell angegeben. Die Übersetzung der Befehle **delay** und **force** wurde nicht in dem Algorithmus eingebaut, da unsere Quellsprache ein erweiterter Lambda-Kalkül gewesen war.

Das angegebene Verfahren realisiert keine vollständige nicht-strikte Semantik der Funktionen. Es handelt sich eher um eine Verzögerung der Berechnung an den Stellen, wo eine unendliche Liste berechnet wird. Das bedeutet, daß die Argumente der Funktionsapplikation berechnet werden. Auf diese Art und Weise ist es möglich, die Evaluation auch anderer Funktionen (nicht nur für die Listenbearbeitung) zu verzögern.

3.4 Behandlung der Typen

In diesem Abschnitt werden zunächst Probleme der Übersetzung der Typen einer einfachen funktionalen Quellsprache in die Typen einer prozeduralen Sprache diskutiert.

Danach führen wir für uns relevante Teile der Typtheorie funktionaler Sprachen ein, und daran anschließend wird ein Algorithmus für die Übersetzung der Typen einer funktionalen Sprache in eine prozedurale Sprache am Beispiel einer Teilsprache von *Scheme* und *Modula-2* angegeben.

3.4.1 Problemstellung

Die Problematik der Übersetzung einer dynamisch typisierten Programmiersprache (wie z. B. *Scheme*) in eine statisch typisierte prozedurale Sprache (wie z. B. *Modula-2*) impliziert in erster Linie das Problem der statischen Herleitung der Typen und Typkonvertierungsoperationen, die während der Laufzeit durchgeführt werden.

Ein Typsystem, bei dem die Typen der Variablen und Ausdrücke hergeleitet werden und bei dem die zur Laufzeit durchzuführenden Typüberprüfungs-Operationen in den Zielcode abgelegt werden, damit die während der Übersetzung nicht korrekt typisierbaren Ausdrücke als typisierbare behandelt werden können, nennen wir *schwaches Typsystem* (engl. *soft type system*). Bei solchen Implementationen gewinnt man an Effizienz bei den vom Compiler erzeugten Programmen (mit der Elimination der zur Laufzeit auszuführenden Typübereinstimungsüberprüfungen).

In [WC94] ist z. B. ein schwaches Typsystem für *Scheme* angegeben. In [HR95] ist beschrieben, wie die dynamisch typisierte Sprache *Scheme* in die statisch typisierte Sprache *ML* übersetzt werden kann, so daß die *Scheme*-Quellprogramme *statisch debugged* werden können. In das Zielprogramm werden die Typkonvertierungsfunktionen abgelegt. Das in [HR95] vorgestellte Typsystem ist das um 5 Regeln erweiterte Standardtypsystem von Hindley-Milner:

1. Typisierung der rekursiven Typen,

2. Typisierung der Applikationen der Typkonvertierungen (*coercions*),

3. Typisierung der Typkonvertierungsabstraktionen,

4. Typisierung der Typkonvertierungsinstanzen und

5. Typisierung der *Scheme*-Definitionen.

Diese Typisierung kann als eine Kombination der schwachen und dynamischen Typisierung betrachtet werden. Das Typsystem ist einerseits schwach, weil einige nach der Typisierung unerlaubte Ausdrücke als korrekt behandelt werden können. Andererseits ist die Typisierung dynamisch, weil die zur *Laufzeit* auszuführenden *coercions* in das Zielprogramm eingebaut werden. In [Jon94] ist gezeigt, wie eine nicht getypte einfache funktionale Programmiersprache in eine *explizit* getypte funktionale Programmiersprache (*polimorpher* Lambda-Kalkül) durch die Typinferenz übersetzt werden kann. In [Feh84] werden die getypten und nicht-getypten Programmiersprachen in ihren allgemeinen Eigenschaften verglichen.

Das zweite Problem, mit dem wir uns auseinandersetzen müssen, ist das Problem des *parametrischen Polimorphismus*. Der angegebene Typprüfungsalgorithmus kann in Typvariablen resultieren. Dies bedeutet, daß der Ausdruck, dessen Typ der Typvariablen entspricht, einen beliebigen Typ haben kann.

In beiden Fällen gilt: Falls ein funktionaler Ausdruck entweder polimorph ist (d. h. sein Typ ist eine Typvariable) oder nicht korrekt getypt werden kann, wird seinem prozeduralen Äquivalent ein *universeller* Typ zugeordnet. Wenn von dem Kontext des Ausdrucks ein konkreter Typ verlangt wird, werden dann an den Stellen Konvertierungsfunktionen aufgerufen.

Unser Typübersetzer entspricht der im Abschnitt 3.3.1 angegebenen Übersetzung einer funktionalen in eine prozedurale Sprache. Aufgrund des einfachen bzw. transparenten Übersetzungsverfahrens erfolgt die Übersetzung der Typen der funktionalen Quellsprache in die Typen der prozeduralen Zielsprache (bei uns ist das die Sprache *Modula-2*) vor der Übersetzung der funktionalen Ausdrücke in die prozedurale Zwischensprache.

Die Informationen über die Typen werden bei der Erzeugung des Zielcodes einer konkreten prozeduralen Sprache (*Modula-2*) gebraucht.

Alle Standardfunktionen der funktionalen Quellsprache haben ihre prozeduralen Äquivalente. Die Typen der Typvariablen, die als Typen der Ergebnisse bzw. Argumente dieser Funktionen auftreten, werden in die den Funktionen des prozeduralen Laufzeitsystems entsprechenden Typen übersetzt.

3.4.2 Typsystem, Typherleitungssystem und Typprüfungssystem

In diesem Abschnit werden die Definitionen der für uns grundlegenden Begriffe der Typtheorie angegeben: Typsystem, Typherleitung, Typprüfung.

Typsystem

Typen bezeichnen Mengen von Werten. Die Typen werden durch die folgende Grammatik beschrieben:

$$\begin{array}{lcll} \mathit{Type} & ::= & \alpha, & \alpha \in \mathit{TypVar} \text{ (Menge der Typvariablen)} \\ & | & H, & H \in \mathit{HTyp} \text{ (Menge der Basistypen)} \\ & | & \beta \to \gamma, & \beta, \gamma \in \mathit{Type} \end{array}$$

Diese Syntaxdefinition setzt zwei Mengen *TypVar* und *HTyp* voraus, die Bezeichnungen (Symbole) für Typvariablen und Basistypen enthalten.

Ein Typsystem ist ein logisches System, mit dessen Hilfe Aussagen über Typen bewiesen werden können. Ein Beweis hat die Form einer Typherleitung (*type inference*), die mit Hilfe der Regeln des Typsystems realisierbar ist. Während für eine logische Ermittlung des Typs bei der theoretischen Behandlung der Typen in einer funktionalen Programmiersprache die Typherleitung, die durch Typregeln durchführbar ist, ausreicht, erfordert die praktische Verwendbarkeit des Typsystems eine automatische Typherleitung. Automatische Typherleitungsysteme nennen wir Typprüfungs- bzw. Typecheckingsysteme.

Bei jedem Typausdruck, der Typvariablen enthält, nehmen wir an, daß alle Typvariablen universell quantifiziert werden. So wird beispielsweise mit dem Typausdruck

$$\alpha \to \beta$$

folgender Typausdruck bezeichnet:

$$\forall\alpha.\forall\beta.\alpha \to \beta$$

Die universelle Quantifizierung erfolgt nur auf der obersten Ebene eines Typausdrucks.[7] Wegen einer einfacheren Schreibweise haben wir in der Grammatik der Typen die Typschemata nicht angegeben. Bevor wir uns den Begriffen Typherleitung und Typprüfung zuwenden, wird noch eine wichtige Notation bei der Typherleitung definiert.

$A \vdash e : \tau$ bedeutet: „Aus der Menge der Annahmen A folgt, daß der Ausdruck e den Typ τ hat." Die Annahme, daß die Variable x den Typ τ hat, wird folgender-

[7] Nach dieser Bedingung sind alle Typen flach (engl. *shallow*), was eine wichtige Bedingung für die Durchführung des Typprüfungsalgorithmus von Milner ist [Mil78].

maßen geschrieben:

$$x : \tau$$

Das Hinzufügen der Annahme $x : \tau$ zur Annahmemenge A, in der es keine dem Typ der Variablen x entsprechende Annahme gibt, wird folgendermaßen bezeichnet:

$$A.x : \tau$$

Die Bezeichnung

$$\frac{A}{B}$$

bedeutet „Aus A kann B abgeleitet werden".
Für die Herleitungsregeln wird noch eine Variablensubstitution angeführt:

$$[\sigma/\alpha]\tau$$

Das bedeutet die Substitution der Variablen α durch die Variable σ im Typausdruck τ, wobei α nicht im Bereich der universell quantifizierten Variablen σ vorkommt.

Zunächst erfolgt eine Erläuterung der Typherleitungsregeln von Cardelli [FH88] und der ihnen entsprechenden Regeln des Typprüfungsalgorithmus von Milner [Mil78].

Typherleitungsregeln

Die in der Abbildung 3.6 angegebenen Regeln ordnen sich in den Typprüfungsalgorithmus von Milner ein.

Dieses System entspricht der logischen Typherleitung, womit zum Beispiel bewiesen werden kann, daß der Ausdruck

$$\textbf{let } f = \lambda x.\ (+\ x\ \mathbf{1})\ \textbf{in } f\ \mathbf{1}$$

einen numerischen Typ hat. Für die automatische Typprüfung soll dieser Algorithmus erweitert werden.

Bei der Typprüfung wird ein für die Typunifikation wichtiger Satz verwendet.

Satz (Robinson, 1965): *Es gibt einen Algorithmus $\mathcal{V}$, der auf ein beliebiges Paar von Ausdrücken σ und τ angewendet wird, so daß $\mathcal{V}(\sigma, \tau)$ entweder erfolgreich ist und eine Substitution U wie folgt liefert:*

(1) $U\ \sigma = U\ \tau$, was bedeutet, daß σ und τ durch U unifiziert werden

(2) bei einer Unifikation von σ und τ durch R gibt es eine Substitution S, so daß $R = SU$

Abbildung 3.6: Typregeln für eine einfache funktionale Sprache-logische Version

Variablen	$A.x : \tau \vdash x : \tau$	$[VAR]$
Bedingte Ausdrücke	$\frac{A \vdash e : truval \;\; A \vdash e' : \tau \;\; A \vdash e'' : \tau}{A \vdash (\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'') : \tau}$	$[COND]$
Abstraktionen	$\frac{A.x : \sigma \vdash e : \tau}{A \vdash (\lambda x.e) : \sigma \to \tau}$	$[ABS]$
Applikationen	$\frac{A \vdash e : \sigma \to \tau \;\; A \vdash e' : \sigma}{A \vdash (ee') : \tau}$	$[APP]$
Let-Ausdrücke	$\frac{A \vdash e' : \sigma \;\; A.x : \sigma \vdash e : \tau}{A \vdash (\mathbf{let}\, x = e'\, \mathbf{in}\, e) : \tau}$	$[LET]$
Der Fixpunkt	$\frac{A.x : \tau \vdash e : \tau}{A \vdash (\mathbf{fix} x.e) : \tau}$	$[FIX]$
Generalisierung	$\frac{A \vdash e : \tau}{A \vdash e : \forall \alpha.\tau}$ (α ist keine freie Variable in A)	$[GEN]$
Spezialisierung	$\frac{A \vdash e : \forall \alpha.\tau}{A \vdash e : [\sigma/\alpha]\tau}$	$[SPEC]$

(3) U involviert nur die in σ und τ vorkommenden Variablen

oder ansonsten $\mathcal{V}(\sigma, \tau)$ nicht erfolgreich ist (es wird keine Substitution geliefert).

Nach diesem Satz ist U der generelle Unifikator.

Beispiele:

$\mathcal{V}(\alpha \to \beta, num \to truval)$ ist durch die Substitution U=[num/α, $truval/\beta$] erfolgreich.

$\mathcal{V}$ $(\alpha \to num, num \to truval)$ kann nicht erfolgreich sein.
$\mathcal{V}(\alpha, \alpha)$ ist erfolgreich durch die identische Substitution I.

Der Typprüfungsalgorithmus $\mathcal{W}$ von Milner

Sei A eine Menge mit Typannahmen und e ein beliebiger Ausdruck. Ist $\mathcal{W}$ erfolgreich, bedeutet die Gleichung

$$\mathcal{W}(A,e) = (T, \tau),$$

daß der Typ des Ausdrucks e τ ist und T eine Substitution darstellt, bei der TA die entsprechenden Typfestsetzungen definiert. Demnach kann diese Beziehung wie folgt definiert werden:

(A) Wenn e eine Variable x ist, dann:

$$T = I \text{ und } x{:}\forall\alpha_1...\alpha_n.\sigma \in \text{A},$$

$\tau = [\beta_1/\alpha_1]...[\beta_n/\alpha_n]\sigma$
wobei $\{\beta_i | 1 \leq i \leq n\}$ neue Variablen sind.

(B) Wenn $e = f\ g$, dann:[8]

$$(R, \rho) = \mathcal{W}(A, f)$$
$$(S, \sigma) = \mathcal{W}(RA, g)$$
$$U = \mathcal{V}(S\rho, \sigma \to \beta).$$

β ist eine neue Variable.

$T = USR$ und $\tau = U\beta$.

[8]Mit $f\ g$ wird eine Funktionsapplikation ohne Klammern bezeichnet.

(C) Wenn $e =$ **if** p **then** f **else** f', dann:

$$(R,\, \rho) = \mathcal{W}(A,\, p)$$
$$\mathrm{U} = \mathcal{V}(\rho,\, truval\,)$$
$$(S,\, \sigma) = \mathcal{W}(URA,\, f)$$
$$(S',\, \sigma') = \mathcal{W}(SURA,\, f')$$
$$U' = \mathcal{V}(S'\sigma,\, \sigma').$$

und

T $= U'S'SUR$ und $\tau = U'\sigma'$

(D) Wenn $e = \lambda x.f$, dann:

$$(R,\, \rho) = \mathcal{W}(A.x{:}\beta,\, f)$$

wobei β eine neue Variable ist.

$T = R$ und $\tau = R\beta \to \rho$.

(E) Wenn $e =$ **fix** $x.f$, dann:

$$(R,\, \rho) = \mathcal{W}(A.x{:}\beta,\, f)$$
$$U = \mathcal{V}(R\beta, \rho)$$

wobei β eine neue Variable ist.

$T = UR$ und $\tau = UR\beta$.

(F) Wenn $e =$ **let** $x = f$ **in** g, dann:

$$(R,\, \rho) = \mathcal{W}(A,\, f)$$
$$(S,\, \sigma) = \mathcal{W}(RA.x{:}\rho',\, g)$$

wobei $\rho' = \forall\alpha_1 ... \alpha_n.\rho$ und $\alpha_1, ..., \alpha_n$ freie Variablen in ρ sind, die in RA nicht vorkommen.

$T = SR$ und $\tau = \sigma$.

Beispiel

Typisierung des Ausdrucks:

let min $= \lambda x.\lambda y.$**if** $(\prec\ x\ y)$ **then** x **else** y
in (min 2 3)

Die erste zu typisierende Sprachkonstruktion ist die **let**-Definition (unter Verwendung von Regel (F) im Algorithmus $\mathcal{W}$):

$\mathcal{W}(e,\ A)=(T,\ \tau)$ $\quad T=SR,\ \tau=\sigma$
$(R,\ \rho)=\mathcal{W}(A,\lambda\ x.\ \lambda\ y.$ **if** $(\prec\ x\ y)$ **then** x **else** $y) = (\ T_1,\ \tau_1)$ $\quad T_1 = R_1, \tau_1 = R_1\beta_1 \to \rho_1$
$(S,\ \sigma)=\mathcal{W}(RA.\text{min:}\rho',\ \text{min 2 3})$
$(R_1,\ \rho_1)=\mathcal{W}(A.x{:}\beta_1,\lambda y.$ **if** $(\prec\ x\ y)$ **then** x **else** $y)$

In vereinfachter Schreibweise:
$\text{A}.x{:}\beta_1 = A_\beta$

$\mathcal{W}(A_\beta,\lambda y.$ **if** $(\prec\ x\ y)$ **then** x **else** $y)= (\ T_2, \tau_2)$ (Typregel D)
$T_2 = R_2,\ \tau_2 = R_2\beta_2 \to \rho_2$
$(R_2,\ \rho_2)= \mathcal{W}(A_\beta.y{:}\beta_2,$**if** $(\prec\ x\ y)$ **then** x **else** $y)$

In vereinfachter Schreibweise:
$A_\beta.y{:}\beta_2 = A_{\beta y}$

$\mathcal{W}(A_{\beta y},$**if** $(\prec\ x\ y)$ **then**x **else** $y) = (\ T_3, \tau_3)$ (Typregel C)
$T_3 = U_3' S_3' S_3\, U_3\, R_3,\ \tau_3 = U_3'\sigma_3'$
$(R_3, \rho_3)= \mathcal{W}(A_{\beta y},\ \prec\ x\ y) = (\ T_4, \tau_4)\ \ T_4 = U_4\, S_4\, R_4,\ \tau_4 = U_4\, \beta_4$

$(R_4\,, \rho_4\,) = \mathcal{W}(A_{\beta y},\ \prec\ x\ )$
$(S_4\,, \sigma_4\,)= \mathcal{W}(R_4\, A_{\beta y}, y) = (I,\ \beta_2)$
$U_4\ = \mathcal{V}(S_4\rho_4, \sigma_4 \to \beta_4)$
$R_4\ = U_5, S_4\ = I, \rho_4\ = num \to truval, \sigma_4\ = \beta_2$

$\mathcal{W}(A_{\beta y},\ \prec\ x\ )=(T_5\,, \tau_5)$ (Typregel B)
$T_5\ = U_5 S_5 R_5\,, \tau_5\ = U_5 \beta_5$
$(R_5\,, \rho_5\,)= \mathcal{W}(A_{\beta y}, \prec) = (I,\ \text{num} \to \text{num} \to \text{truval})$
$R_5\ = I,\ \rho_5\ = num \to num \to truval$
$(S_5\,, \sigma_5\,)= \mathcal{W}(R_5\, A_{\beta y}, x) = (I, \beta_1\,)$
$S_5\ = I,\ \sigma_5\ = \beta_1$
$U_5 = \mathcal{V}(\rho_5, \sigma_5\ \to\ \beta_5)\ =\ \mathcal{V}\ (num\ \to\ num\ \to\ truval, \beta_1\ \to\ \beta_5)\ =\ (num/\beta_1, num\ \to\ truval/\beta_5)$
$T_5\ = U_5, \tau_5 = num \to truval$

Nun können schrittweise die Werte der Variablen R und ρ durch bekannte Typvariablen ausgedrückt werden.

$R_4\ = U_5, \rho_4\ = num \to truval, U_4\ = (num/\beta_2, truval/\beta_4\,)$

$T_4\ = U_4\, U_5, \tau_4\ = truval, R_3\ = U_4\, U_5, \rho_3\ = truval$
$U_3\ = \mathcal{V}(\rho_3, truval) = I$
$(S_3\,, \sigma_3\,)=\mathcal{W}(\,U_3\, R_3\, A_{\beta y}, x)=(I,\ \text{num})$

$(S_3', \sigma_3') = \mathcal{W}(S_3\, U_3\, R_3\, A_{\beta y}, y) = (I, \text{num})$
$U_3' = \mathcal{V}(S_3'\sigma_3, \sigma_3')$
$S_3 = I, S_3' = I, \sigma_3 = num, \sigma_3' = num,\ U_3' = \mathcal{V}(\text{num, num}) = I$

$R_2 = U_4\, U_5,\ T_2 = R_2 = U_4\, U_5, \rho_2 = num, \tau_2 = U_4\, U_5 \beta_2 \to num = num \to num$

$R_1 = T_2 = U_4\, U_5, \rho_1 = \tau_2 = num \to num,$
$T_1 = R_1 = U_4\, U_5, \tau_1 = R_1 \beta_1 \to \rho_1 = num \to num \to num, R = T_1 = U_4\, U_5,$
$\rho = \tau_1 = num \to num \to num$

Die Werte der Variablen S und σ berechnen wir folgendermaßen

$(S, \sigma) = \mathcal{W}(RA_{\beta y}.min : \rho', \text{min 2 3}) = (T_{10}, \tau_{10})$

In vereinfachter Schreibweise:
$A_{\beta y \rho} = A_{\beta y}.min : \rho'$

$\rho' = \forall \alpha_1 ... \alpha_n . \rho$ wobei $\alpha_1 ... \alpha_n$ freie Variablen in ρ, die nicht in $RA_{\beta y}$ vorkommen, sind.

$(S, \sigma) = \mathcal{W}(RA_{\beta y \rho},\ min\ 2\ 3) = (T_{10}, \tau_{10})$ (Regel B)
$T_{10} = U_{10} S_{10} R_{10},\ \tau_{10} = U_{10} \beta_{10}$
$(R_{10}, \rho_{10}) = \mathcal{W}(RA_{\beta y \rho}, \text{min 2})$
$(S_{10}, \sigma_{10}) = \mathcal{W}(\ R_{10} RA_{\beta y \rho}, 3) = (I, \text{num})$
$U_{10} = \mathcal{V}(S_{10} \rho_{10}, \sigma_{10} \to \beta_{10})$
$S_{10} = I, \sigma_{10} = num, T_{10} = U_{10} S_{10} R_{10}, \tau_{10} = U_{10} \beta_{10}$
$(R_{10}, \rho_{10}) = \mathcal{W}(RA_{\beta y \rho}, \text{min 2}) = (T_{11}, \tau_{11}),\ T_{11} = U_{11} S_{11} R_{11},\ \tau_{11} = U_{11} \beta_{11}$

$(R_{11}, \rho_{11}) = \mathcal{W}(RA_{\beta y \rho}, min) = (I, num \to num \to num)$ (Regel A)
$(S_{11}, \sigma_{11}) = \mathcal{W}(R_{11} RA_{\beta y \rho}, 2) = (I, \text{num})$ (Regel A)

Nun werden wieder unbekannte Variablen durch bekannte ausgedrückt:

$U_{11} = \mathcal{V}(S_{11} \rho_{11}, \sigma_{11} \to \beta_{11})$
$R_{11} = \text{I},\ \rho_{11} = \text{num} \to \text{num} \to \text{num},\ S_{11} = I,\ \sigma_{11} = \text{num},\ U_{11} = (\text{num} \to \text{num} \,/\, \beta_{11}),$
$R_{10} = T_{11},\ \rho_{10} = \text{num} \to \text{num},$
$U_{10} = \mathcal{V}(\ \text{num} \to \text{num, num} \to \beta_{10}),$

$U_{10} = (num/\beta_{10}), T_{10} = U_{10}\, T_{11}, \tau_{10} = num, S = U_{10}\, T_{11}, \sigma = num.$

Das abschließend hergeleitete Tupel des Typs des Ausdrucks ist (T, τ), wobei:
$T = U_{10}\, T_{11}\, U_4\, U_5, \tau = num.$

Wir haben die aus unserer Sicht wesentlichen Begriffe und Zusammenhänge der Typtheorie eingeführt. Der folgende Abschnitt bezieht sich auf die Probleme der

optimalen Zuordnung der Typen an die Variablen bzw. Prozeduren der Zielsprache bei einer Transformation zwischen funktionalen und prozeduralen Programmiersprachen.

Die konkrete Implementation des Verfahrens basiert auf einer Modifikation des Typprüfungalgorithmus von *Milner*. Im folgenden definieren wir den Typübersetzer.

3.4.3 Der Typübersetzer: Modifizierter Algorithmus $\mathcal{W}$

Unseren Typübersetzer definieren wir durch eine Erweiterung des Algorithmus $\mathcal{W}$. Die Erweiterungen beziehen sich auf die grundlegenden Typen, die Menge der Typannahmen und die Unifikations- bzw. Substitutionsfunktionen. Der Typübersetzer soll auch die Typvariablen übersetzen. Das Verfahren dieser Übersetzung wird auch hier definiert.

Im folgenden werden die o.g. Komponenten des Typübersetzers beschrieben.

Grundlegende Typen

Definieren wir drei grundlegende Typen: **se**, **truval**, **num** (wir befinden uns auf der Ebene der prozeduralen Zielsprache) sowie einen besonderen Typ **bot**, der als Indikator einer erfolglosen Typisierung genutzt wird.

Jeder Typ hat seine Priorität: **se** mit der größten Priorität, gefolgt von **num** und **truval**, sowie **bot** mit der kleinsten Priorität.

Für die drei eingeführten Haupttypen unserer Sprache definieren wir folgende ,,Quasitypen":

$$\mathbf{se}_{num},\ \mathbf{se}_{truval},\ \mathbf{num}_{se},\ \mathbf{truval}_{se}$$

Diese Typen sind Compilerdirektiven für die Typkonvertierung und erscheinen, wenn im funktionalen Code Polimorphismus auftritt. Die Notation bedeutet folgendes:

$\mathbf{se}_{num}$ - der Typ des funktionalen Ausdrucks ist **se**, aber im Kontext wird ein numerischer Typ erwartet

$\mathbf{se}_{truval}$ - der Ausdruck hat den Typ **se**, aber im Kontext wird der logische Typ erwartet

$\mathbf{num}_{se}$ - der Ausdruck hat den numerischen Typ, aber im Kontext wird der Typ **se** erwartet

$\mathbf{truval}_{se}$ - der Ausdruck hat den logischen Typ, aber im Kontext wird der Typ **se** erwartet. Jedem ,,Quasityp" entspricht dabei der Typ, der sich in seinem Index befindet. Z. B. entspricht dem Quasityp $\mathbf{se}_{truval}$ der Typ **truval**.

Typannahmen

Die Menge der Typannahmen entspricht den tatsächlichen Gegebenheiten, auf die sich die Typdeduktion stützt. Eine derartige Menge wird während der laufenden Übersetzung an die neu deduzierten Typen angepaßt. Zu einem Typisierungsmodell gehören anfänglich Typen aller Konstanten bzw. Standardfunktionen der Sprache. In unserer Implementation gibt es drei grundlegende Typen für die Funktionen des Laufzeitsystems: **num**, **truval** und **se** (mit denen die Funktionskonstruktoren definiert werden können).

Die von uns zugrunde gelegte Quellsprache besteht aus erweiterten Lambda-Ausdrücken.

Die in der Abbildung 3.7 vorgestellte Tabelle bietet einen Überblick über die Typen der Sprachelemente der von uns betrachteten Teilsprache von *Scheme*. Die Bezeichnung α steht für einen beliebigen Typ und $[\alpha]$ ist eine Liste, deren Elemente einen beliebigen Typ haben können.

Alle in der Tabelle genannten Funktionen haben ihre Äquivalente in unserem prozeduralen Modell, die entweder ein Sprachelement der prozeduralen Sprache sind oder zum Laufzeitsystem des Compilers gehören. In der Tabelle in der Abbildung 3.8 sind die prozeduralen Äquivalente der Konstanten bzw. Funktionen mit ihren Typen angegeben. Das prozedurale Äquivalent einer Funktion f wird mit f_p bezeichnet. Der Typ se ist der Typ der S-Ausdrücke (S-Expressions).

In *Modula-2* entsprechen die Typen **truval** und **num** den Standardtypen BOOLEAN und INTEGER. Der Typ *se* ist dagegen ein Typ des Laufzeitsystems, der S-Ausdrücke (S-expressions) nachbildet. In *Modula-2* sind viele der angeführten Funktionen bereits Standardfunktionen: or_p, and_p, not_p, $+_p$, $-_p$, $*_p$ ... Die restlichen Funktionen werden im Laufzeitsystem definiert.

Unifikation und Substitution

Der Typ eines funktionalen Ausdrucks kann auf zwei Arten ermittelt werden:

- Der ,,von außen" deduzierte Typ - *the Type deduced from outside* [Jon87]:
  Dieser Typ ist das Resultat der Typanalyse des Ausdruckskontextes.

- Der ,,von innen" deduzierte Typ - *the type deduced from inside*:
  Dieser Typ soll bei der Analyse des Ausdrucks ermittelt werden.

In unserem System kann es zu Unterscheidungen der Übersetzung der zwei o.g. Typen kommen, obwohl es sich auf der Ebene der funktionalen Sprache um zwei gleiche Typen handelt. Sei beispielsweise ein Ausdruck in der Sprache *Scheme* folgendermaßen definiert:

```
(letrec (( a (quote (1 2 3)))) (+ (car a) 1))
```

Abbildung 3.7: Sprachelemente der funktionalen Quellsprache mit ihren Typen

Standardwerte	*#t*: truval
	#f: truval
Standardprozeduren	*eqv?*: $(\alpha, \alpha) \rightarrow$ truval
	eq?: $(\alpha, \alpha) \rightarrow$ truval
	equal?: $(\alpha, \alpha) \rightarrow$ truval
	pair?: $\alpha \rightarrow$ truval
	list?: $\alpha \rightarrow$ truval
	cons: $(\alpha, [\alpha]) \rightarrow [\alpha]$
	car: $[\alpha] \rightarrow \alpha$
	cdr: $[\alpha] \rightarrow [\alpha]$
	null?: $\alpha \rightarrow$ truval
	length: $[\alpha] \rightarrow$ num
	append: $([\alpha], [\alpha]) \rightarrow [\alpha]$
	reverse: $[\alpha] \rightarrow [\alpha]$
	member: $(\alpha, [\alpha]) \rightarrow$ truval
	symbol?: $\alpha \rightarrow$ truval
	or, *and*: (truval, truval) $\rightarrow$ truval
	not: truval $\rightarrow$ truval
Numerische Operationen	$+$,-,$*$, $/$: (num, num) $\rightarrow$ num
	quotient, *remainder*, *modulo* : (num, num) $\rightarrow$ num
	$=, \prec, \succ, \prec=, \succ=$: (num, num) $\rightarrow$ truval

Abbildung 3.8: Prozedurale Äquivalente der Sprachelemente der funktionalen Quellsprache und ihre Typen

Standardwerte	*true*: truval *false*:truval
Standardprozeduren	$eqv?_p$: (se , se) → truval $eq?_p$: (se, se) → truval $equal?_p$: (se, se) → truval $pair?_p$: se → truval $list?_p$: se → truval $cons_p$: (se, se) → se car_p: se → se cdr_p: se → se $null?_p$: se → truval $list_p$: se → truval $length_p$: se → num $append_p$: (se, se) → se $reverse_p$: se → se $member_p$: (se, se) → truval $symbol?_p$: (se, se) → truval or_p, and_p: (truval, truval) → truval
Numerische Operationen	$+_p$,$-_p$,$*_p$, $/_p$: (num, num) → num $quotient_p$,$remainder_p$, $modulo_p$: (num, num) → num $=_p$, $\prec_p$, $\succ_p$, $\prec=_p$, $\succ=_p$: (num, num) → truval

Die Operation + ist korrekt angewendet, weil a eine Liste von natürlichen Zahlen ist. In unserem Modell ist a eine Liste, die durch den Typ **se** repräsentiert wird und *(car a)* ein Knoten, der ebenfalls einen Typ **se** hat. Soll diese Struktur als eine Zahl behandelt werden, stellt dieser Knoten eine *boxed*-Repräsentation der betreffenden Zahl dar, wobei Konvertierungsfunktionen aufgerufen werden. Die Übersetzung des Ausdrucks *(+ (car a) 1)* lautet:

ValIntSE(HeadSE(a)) + 1

ValIntSE ist dabei eine Konvertierungsfunktion. Demnach muß die Unifikation die eingebauten Datenstrukturen als uneingebaute Strukturen behandeln und als Nebeneffekt die Konvertierungsfunktionen aufrufen. Das ist eine übliche Behandlung des Polimorphismus. Listen werden meistens bei den statischen Typsystemen funktionaler Programmiersprachen auf andere Art und Weise behandelt: Eine Listenfunktion wird als eine polimorphe Funktion definiert. Z. B. hätte die Funktion **car** die folgende Funktionalität:

$\mathbf{car} : [\alpha] \rightarrow \alpha$

Falls beispielsweise eine numerische Liste als Argument der Funktion *car* auftritt, wird die Typvariable α mit dem numerischen Typ unifiziert. Diese Unifizierung gibt uns Auskunft, wie die Elemente der Liste behandelt werden dürfen (in unserem Fall: als numerische Typen). Hier haben wir uns nur auf den Typ **se** beschränkt, weil die *Scheme*-Listen Objekte verschiedener Typen sein können. Falls der Kontext einer Liste einen beliebigen Typ verlangt, werden einfach die Typkonvertierungsfunktionen, die aus dem Listenknoten (auf der Implementationsebene) ein Objekt dieses Typs „herausnehmen", aufgerufen.

In unserem Modell werden im Fall eines polimorphen Programmverhaltens mit Hilfe der Substitution dem Compiler durch entsprechende Direktiven die Art der ermittelten Politypen übergeben. Hauptidee dabei ist die Definition der Compilerdirektiven durch „Quasitypen", die während der Typanalyse des Programms durch Substitution produziert werden. So wird die Information über die Konvertierungsfunktion *ValIntSE* im Programmteil

ValIntSE(HeadSE(a)) + 1

durch den Quasitypen se_{num} ermittelt, der dem Ausdruck

(car a)

entspricht.

Nach den oben angeführten Voraussetzungen (Satz von Robinson) wird die Substitutionsfunktion durch einen Unifikationsalgorithmus $\mathcal{V}$ definiert. Also:

$$\mathcal{V}\text{: } \mathit{Type} \times \mathit{Type} \rightarrow \mathit{substitution}\ .$$

Die Unifikation

$$\mathcal{V}(\alpha \rightarrow \beta,\ \mathit{num} \rightarrow \mathit{truval})$$

liefert beispielsweise die Substitution

$$\mathrm{U}=[\mathit{num}/\alpha,\ \mathit{truval}/\beta]\ .$$

Die semantische Bedeutung einer Substitution liegt entweder darin, einer Typvariablen ihre entsprechenden Typen zu übermitteln oder eine Überprüfung der Typübereinstimmung durchzuführen. Jede Typvariable bzw. jeder Typ, welche als Argumente der Substitutionsfunktion auftreten, verfügen über ihre entsprechenden Ausdrücke. Unerlaubte Unifikationen geben uns dabei über die Art der Typkollisionen und Ausdrücke Auskunft. Unser Modell verfügt über eine solche Unifikation, welche für eine Typänderung (Typ $\rightarrow$ Quasityp) ursächlich ist, wobei der neue Typ (Quasityp) auf der Ebene der Übersetzung ein Zeichen für den Aufruf der Konvertierungsfunktion ist. Für die Umsetzung unserer Idee wird folgendes definiert:

Einzelnunifikationen sind Unifikationen auf der niedrigsten Ebene. Beispielsweise werden bei der Unifikation

$$\mathcal{V}(\mathbf{se} \rightarrow \mathbf{num},\ \mathbf{num} \rightarrow \mathbf{truval})$$

zwei Einzelnunifikationen durchgeführt: $\mathbf{v}_e(\mathbf{se}, \mathbf{num})$ und $\mathbf{v}_e(\mathbf{num},\mathbf{truval})$. Definieren wir nun zwei Einzelnunifikationen mit besonderer Priorität des ersten bzw. des zweiten Argumentes:

$$v_1(\mathbf{se}, \mathbf{num}) = v(\mathbf{se},\ \mathbf{num}_{se})$$
$$v_1(\mathbf{se}, \mathbf{truval}) = v(\mathbf{se},\ \mathbf{truval}_{se})$$
$$v_1(\mathbf{num}, \mathbf{se}) = v(\mathbf{num},\ \mathbf{se}_{num})$$
$$v_1(\mathbf{truval}, \mathbf{se}) = v(\mathbf{truval},\ \mathbf{se}_{truval}).$$

In allen anderen Fällen gilt: $v_1 = v$.

$$v_2(\mathbf{se},\ \mathbf{num}) = v(\mathbf{se}_{num},\ \mathbf{num})$$
$$v_2(\mathbf{se}, \mathbf{truval}) = v(\mathbf{se}_{truval},\ \mathbf{truval}\ )$$
$$v_2(\mathbf{num}, \mathbf{se}) = v(\mathbf{num}_{se},\ \mathbf{se})$$

$$v_2(\mathbf{truval}, \mathbf{se}) = v(\mathbf{truval}_{se}, \mathbf{se}\ ).$$

In allen anderen Fällen gilt: $v_2 = v$.

Die Unifikationsfunktionen, denen die Einzelnunifikationen v_1 bzw. v_2 entsprechen, werden mit $\mathcal{V}_1$ bzw. $\mathcal{V}_2$ bezeichnet.
Aufgrund der Typenpriorität ist noch eine zusätzliche Funktion für die Typunifikation erforderlich:

$$\mathcal{V}_p(typ_1, typ_2) = \begin{cases} \mathcal{V}_1(typ_1, typ_2), & falls\ typ_1\ \ größere\ Priorität\ als\ typ_2\ hat; \\ \mathcal{V}_2(typ_1, typ_2), & falls\ typ_2\ \ größere\ Priorität\ als\ typ_1\ hat; \\ \mathcal{V}(typ_1, typ_2), & andernfalls. \end{cases}$$

Die Behandlung der Funktionen höherer Ordnung wird im folgenden Abschnitt beschrieben.

Übersetzung der Typvariablen

Die Typisierung eines Ausdrucks nach dem $\mathcal{W}$-Algorithmus kann auch von Typvariablen abhängen. Das bedeutet, daß der Ausdruck keinen deduzierbaren konkreten Typ besitzt. Typvariablen als Ergebnis der Typisierung zeigen uns den Polimorphismus des Ausdrucks. Wie oben ausgeführt, muß die Übersetzung ein typisiertes Programm in *Modula-2* liefern. Solche Ausdrücke typisieren wir mit dem Typ **se**, weil es eine Möglichkeit der Konvertierung dieses Typs in alle andere Typen gibt. Zum Beispiel hat der Typ des K-Kombinators (Kanzellator-Funktion)

```
(define k_comb
  (lambda (c x)
     c))
```

die folgende Form:

$$(\alpha, \beta) \rightarrow \alpha$$

In unserem Modell wird der Typ dieser Funktion folgendermaßen übersetzt:

$$(\mathbf{se}, \mathbf{se}) \rightarrow \mathbf{se}$$

Auswahl des Variablentyps

Die Variablen verfügen über keinen „inneren" Typ, sondern werden lediglich nach ihrem Kontext typisiert. Unsere Idee besteht darin, die Informationen über den Typ

einer Variablen zu sammeln und ihr den Typ mit der größten Priorität zuzuordnen.

Im Falle eines Substitutionskonflikts wird der Variablen der Typ mit der größten Priorität zugeordnet, alle anderen Substitutionen, die mit dieser Typisierung nicht übereinstimmen, werden modifiziert.

Wenn α eine Variable mit einem verschiedenartig deduzierbaren Typ und *alphaTyp* der Typ dieser Variablen mit der größten Priorität ist, werden diejenigen Komponenten der Substitution, die sich auf α beziehen, folgendermaßen geändert:

$$einTyp/\alpha \rightarrow alphaTyp_{einTyp}/\alpha$$

Alle Änderungen dieser Art haben ihre entsprechenden Vorkommen dieser Variablen, die weiter im Compilersystem mit Konvertierungsfunktionen ergänzt werden.

Der $\mathcal{W}_{smc}$-Algorithmus

Definieren wir nun den $\mathcal{W}_{smc}$-Algorithmus

$$\mathcal{W}_{smc}(A,e) = (T, \tau)$$

wobei:

(I) Wenn e eine Variable x ist, dann gilt:
$T = I$ und $x{:}\forall\alpha_1...\alpha_n.\sigma \in$ A,

$\tau = [\beta_1/\alpha_1]...[\beta_n/\alpha_n]\sigma$
wobei $\{\beta_i | 1 \leq i \leq n\}$ neue Variablen sind.

(II) Wenn $e = f\ g$, dann gilt:

$$(R, \rho) = \mathcal{W}_{smc}(A, f)$$
$$(S, \sigma) = \mathcal{W}_{smc}(RA, g)$$
$$U = \mathcal{V}_1(S\rho, \sigma \rightarrow \beta)$$

β ist eine neue Variable.

$T = USR$ und $\tau = U\beta$.

(III) Wenn $e =$ **if** p **then** f **else** f', dann gilt:

$$(R, \rho) = \mathcal{W}_{smc}(A, p)$$
$$\mathrm{U} = \mathcal{V}_2(S\rho, truval\ )$$
$$(S, \sigma) = \mathcal{W}_{smc}(URA, f)$$
$$(S', \sigma') = \mathcal{W}_{smc}(SURA, f')$$
$$U' = \mathcal{V}_p(S'\sigma, \sigma').$$

Dabei gilt

T = *U'S'SUR* und τ = *U'σ'*

(IV) Wenn $e = \lambda x.f$, dann gilt:

$$(R,\ \rho) = \mathcal{W}_{smc}(A.x{:}\beta,\ f)$$

β ist eine neue Variable.

$T = R$ und $\tau = R\beta \to \rho$.

(V) Wenn $e =$ **fix** $x.f$, dann gilt:

$$(R,\ \rho) = \mathcal{W}_{smc}(A.x{:}\beta,\ f)$$
$$U = \mathcal{V}_2(R\beta, \rho).$$

β ist eine neue Variable, $T = UR$ und $\tau = UR\beta$.

(VI) Wenn $e =$ **let** $x = f$ **in** g, dann gilt:

$$(R,\ \rho) = \mathcal{W}_{smc}(A,\ f)$$
$$(S,\ \sigma) = \mathcal{W}_{smc}(RA.x{:}\rho',\ g)$$

wobei $\rho' = \forall\alpha_1 ... \alpha_n.\rho$ und $\alpha_1, ..., \alpha_n$ freie Variablen in ρ sind, die in RA nicht vorkommen.

$T = SR$ und $\tau = \sigma$.

Beispiel

Übersetzen wir den Typ des folgenden Ausdrucks:

$e = \lambda\ x.$ **cons** x (+ x **1**)

$\mathcal{W}_{smc}(A,\ e) = (T,\ \tau)$ mit $T = R$, $\tau = R\beta \to \rho$

$A_\beta = A.x{:}\ \beta$

$(R,\ \rho) = \mathcal{W}_{smc}(\ A_\beta,$ **cons** x (+ x **1**)) $= (T_1,\ \tau_1)$

$T_1 = U_1S_1R_1$, $\tau_1 = U_1\beta_1$

$(R_1,\ \rho_1) = \mathcal{W}_{smc}(A_\beta,$ **cons** $x) = (T_2,\ \tau_2)$
$(S_1,\ \sigma_1) = \mathcal{W}_{smc}(R_1A_\beta,$ + x **1**$) = (T_3,\ \tau_3)$
$U_1 = \mathcal{V}_1(S_1\rho_1,\ \sigma_1 \to \beta_1)$

$T_3 = U_3 S_3 R_3$
$(R_3, \rho_3) = \mathcal{W}_{smc}(A\beta, + x) = (T_4, \tau_4)$
$T_4 = U_4 S_4 R_4$
$(R_4, \rho_4) = \mathcal{W}_{smc}(A\beta, +)$
$(S_4, \sigma_4) = \mathcal{W}_{smc}(R_4 A_\beta, \mathbf{1})$

Alle weitere Typdeduktionen basieren auf der Typisierung der Funktionsapplikation, wobei wir an dieser Stelle nur die Besonderheiten des Beispiels angeben (vgl. Beispiel aus 3.4.2):

...

$U_4 = (\mathbf{num} \to \mathbf{num} \to \mathbf{num}, \beta \to \beta_4) = [\mathbf{num}/\beta, \mathbf{num} \to \mathbf{num}/\beta_4]$

...

$U_2 = [\mathbf{se}/\beta, \mathbf{se} \to \mathbf{se}/\beta_2]$

...

$\tau = R\beta \to \rho$
$T = R = T_1 = U_1 U_2 U_3 U_4 =$
$=[... \mathbf{num}/\beta, \mathbf{num} \to \mathbf{num}/\beta_4, \ldots, \mathbf{se}/\beta, \mathbf{se} \to \mathbf{se}/\beta_2, \ldots]$

Nach der Regel für die Auswahl des Typs einer Variablen (bei uns ist das die Variable x, deren Typ mit β bezeichnet ist) transformieren wir die Substitution folgendermaßen:

$$[... \mathbf{se}_{num}/\beta, \mathbf{num} \to \mathbf{num}/\beta_4, \ldots, \mathbf{se}/\beta, \mathbf{se} \to \mathbf{se}/\beta_2, \ldots]$$

β hatte zwei deduzierte Typen - **se** und **num**. Der Typ mit der größten Priorität ist **se**, weshalb wir die Transformation

$$\mathbf{num}/\beta \Leftrightarrow\!\!\!\!\rightarrow \mathbf{se}_{num}/\beta \qquad (*)$$

durchgeführt haben.

An dieser Stelle wird im Compiler eine Konvertierungsfunktion aufgerufen. Dieser Typ der Variablen x entspricht der Kontextanalyse des Vorkommens der Variablen x als Argument der Funktion $+$.

Analysieren wir noch die Substitution U_1:

$$U_1 = \mathcal{V}_1(\mathbf{se} \to \mathbf{se}, \mathbf{num} \to \beta_1)$$
$$= \mathcal{V}(\mathbf{se} \to \mathbf{se}, \mathbf{num}_{se} \to \beta_1) \text{ (nach der Definition von } \mathcal{V}_1) \qquad (**)$$

Der Typ der Funktion ist: $\mathbf{se} \to \mathbf{se}$.

Geben wir jetzt noch die transformierte Funktion in *Modula-2* an:

```
PROCEDURE F(X: SExp): SExp;
BEGIN
  RETURN ConsSE (X, QIntSE((ValIntSE(X) + 1)))
END F;
```

Der Aufruf der Konvertierungsfunktion **ValIntSE** ist durch die Substitutionstransformation (*) und der Aufruf von **QIntSE** durch die Transformation (**) realisiert.

Wir haben gezeigt, wie der Typprüfungalgorithmus von Milner für die Erzeugung des Zielcodes mit einfachen Typen (wenn es möglich ist) benutzt werden kann. Dazu wurden neue Unifikationsfunktionen und Substitutionen definiert. Die Zuordnung der Typen der Variablen und Funktionen des Zielcodes ist übereinstimmend mit den im Kap. 3 definierten Übersetzungsregeln und Ergebnis der Analyse des funktionalen **Quellcodes**.

In [Hal94] ist z. B. beschrieben, wie die Hindley-Milners-Typinferenz zwecks Optimierung der Listenrepräsentationen benutzt worden ist: Zwei Listenrepräsentationen wurden eingeführt - ,, normale" und ,,optimale". Die Funktionen für die Konvertierung einer in die andere Listenrepräsentation werden mit Hilfe des Systems für die Typinferenz aufgerufen.

In [HR95] ist ein Typsystem mit Konvertierungsfunktionen zwecks Übersetzung der Sprache *Scheme* in die Sprache *ML* angegeben. Im Verfahren des Einbaus der Typkonvertierungsfunktionen wird am Anfang jeder Ausdruck des Quellprogramms mit einer zusätzlichen Tykonvertierungsfunktion erweitert, die *coercions*-Signaturen werden entsprechend der statischen Typisierungsregeln unifiziert. Weiterhin wird die bestmögliche Lösung für die (Elimination der) Aufrufe der Typkonvertierungsfunktionen gesucht (durch *value flow graph* usw.).

3.4.4 Funktionen als Objekte erster Ordnung

In diesem Abschnitt wird das Problem der Typübereinstimmung der Ausdrücke, in denen Funktionen als Objekte erster Ordnung erscheinen, analysiert. Falls die Funktionen in einer monomorphen Form vorkommen und genauso behandelt werden, ist es klar, daß sie mit Hilfe des definierten Typübersetzungsystems korrekt übersetzt werden können. Die Typübereinstimmung der Funktionen, deren innerer Typ sich vom äußeren Typ unterscheidet, wird mit Hilfe der neu definierten semantisch äquivalenten typübereinstimmenden Funktionen (deren Typ in Übereinstimmung mit dem Typ ihres Kontextes ist) erreicht. Darüber hinaus muß die Ordnung jeder Funktion statisch ableitbar sein, d. h. die Typvariablen in den Funktionstypen werden mit Basistypen unifiziert. Betrachten wir das folgende funktionale Programm:

```
(define zweimal
```

```
(lambda (f x)
   (f (f x))))

(define plusplus
 (lambda (x)
   (+ x x)))

(define main (zweimal plusplus 3))
```

Der Typ der Funktion *zweimal* ist:

$$(\alpha \to \alpha, \alpha) \to \alpha$$

Die Funktion *plusplus* ist monomorph und wird als Argument der (polimorphen) Funktion *zweimal* verwendet.

Nach unserem Algorithmus wird der Typ der Funktion *zweimal* folgendermaßen übersetzt:

$$(\mathbf{se} \to \mathbf{se}, \mathbf{se}) \to \mathbf{se}$$

Der Aufruf der Funktion *zweimal* mit der Funktion *plusplus* (deren Typ ist: **num** $\to$ **num**) führt zu einem Typübereinstimmungfehler. Deswegen wird eine neue Funktion *plusplusA* in der prozeduralen Zielsprache definiert:

```
PROCEDURE plusplusA(x: SExp): SExp
BEGIN
  QIntSE(plusplus(ValIntSE(x)))
END plusplusA;
```

Die Funktion *zweimal* kann nun mit der Funktion *plusplusA* aufgerufen werden. Allgemein wird folgendes Prinzip realisiert: Für jeden Typübereinstimmungsfehler einer beliebigen Funktion *f*, die als ein Objekt erster Ordnung behandelt wird, wird eine neue Funktion definiert (in o.g. Beispiel war das die Funktion *plusplusA*), die dem Typ nach dem betrachteten Kontext von *f* entspricht. Statt der Funktion *f* wird die neudefinierte Funktion in dem Programm abgelegt.

Eine ähnliche Idee ist in [Ler94] beschrieben. Auf einer derartigen Behandlung der polimorphen Funktionen basiert auch der *SML* of New Jersey-Compiler [SA95].

3.5 Implementation der Transformationsstrategie

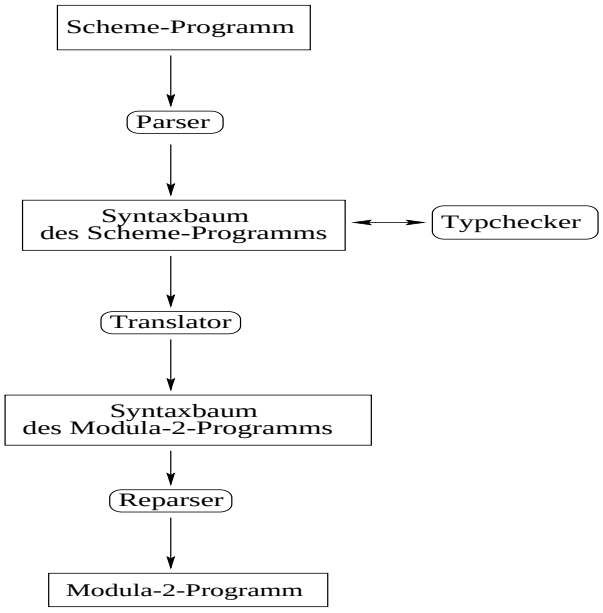
In diesem Abschnitt wird ein Überblick über die Implementation des Compilersystems gegeben. Dabei werden wir zunächst auf die grundlegende Struktur des Compilers eingehen und anschließend zwei seiner Komponenten näher beschreiben.

Der Compiler selbst ist in *Modula-2* implementiert und besteht aus 13 Modulen. Die Auswahl von *Modula-2* als Implementationssprache wurde deshalb vorgenommen, um ein portables System zu realisieren: Wenn die Zielsprache des Compilers *Modula-2* ist, muß zur Weiterbearbeitung der erzeugten Programme ein *Modula-2*-System zur Verfügung stehen, mit dem auch der *Scheme-Modula-2*-Compiler selbst verarbeitet werden kann. Damit hängt der Compilationsprozeß lediglich von einem Compilersystem (für *Modula-2*) ab.

Struktur des Übersetzungsprozesses

In der Abbildung 3.9 ist das Übersetzungsprozeß dargestellt.

Abbildung 3.9: Übersetzungsprozeß des *SMC*-Compilers



Die wichtigsten Komponenten des Compilers sind der Abbildung zu entnehmen: Parser, Translator, Typchecker, Reparser.

Die Eingabe des Compilers ist ein Programm in der von uns definierten Teilsprache von *Scheme*. Die *Scheme*- und *Modula-2*-Programme werden intern durch Listen repräsentiert. Diese bilden eine Form der Darstellung von Syntaxbäumen der Pro-

gramme. Dabei überführt der Parser ein *Scheme*-Programm in den entsprechenden Syntaxbaum.

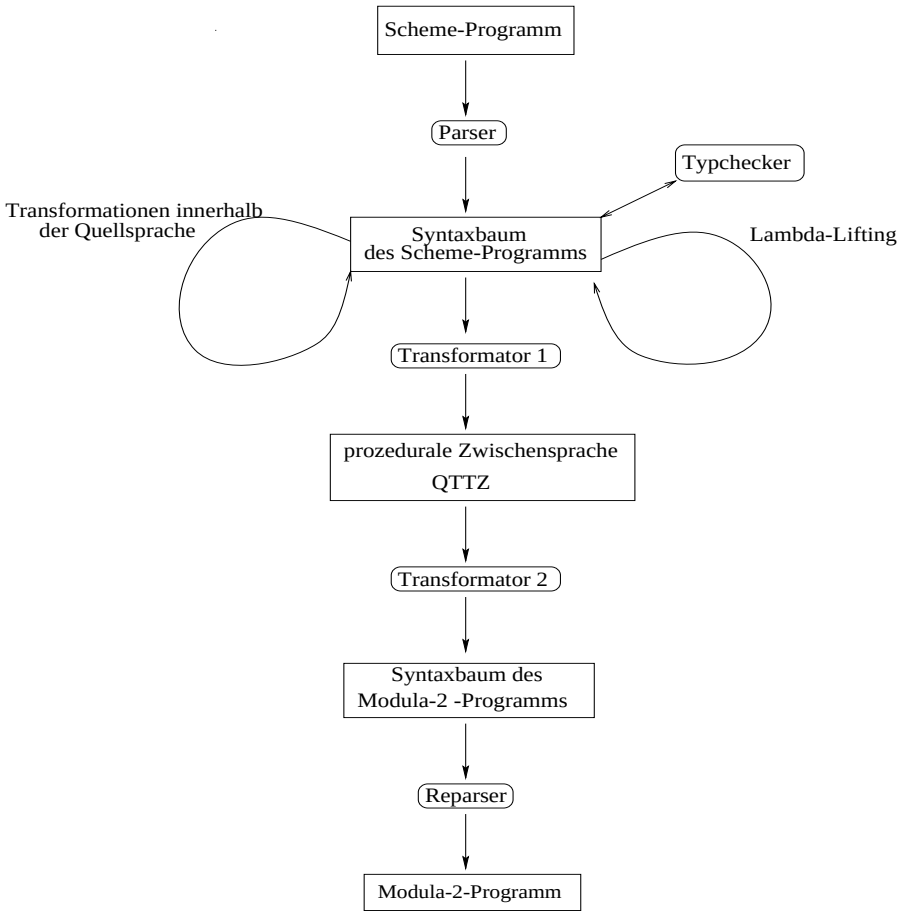
Die Übersetzung in den Syntaxbaum des *Modula-2*-Programms wird durch den Translator ausgeführt. Die Typherleitung und der Transformationsprozeß laufen gleichzeitig: Der Translator ruft den Typchecker auf, der Typchecker leitet den Typ her und gibt ihn an den Translator zur weiteren Bearbeitung. Die Ausgabe des Translators - ein *Modula-2*-Baum - wird mit Hilfe eines Reparsers in Form eines *Modula-2*-Programms in die Zieldatei ausgegeben. Abbildung 3.9 gibt die Grobstruktur des Compilers wieder. Unter Berücksichtigung der einzelnen Transformationsschritte, wie im Abschnitt 3.3 beschrieben, ergibt sich die in Abbildung 3.10 dargestellte Feinstruktur des Compilers.

Aufbau des Translators

Nach der syntaktischen Analyse des *Scheme*-Programms und seiner Überführung in den entsprechenden Syntaxbaum beginnt der Prozeß der Transformation durch den Translator. Jede *Scheme*-Konstruktion hat eine ihr entsprechende Transformationsregel bzw. eine Transformationsprozedur. Die Transformationsprozeduren werden in der Hauptprozedur des Translators aufgerufen. Beispielsweise sieht der Teil der Hauptprozedur des Translators für die Behandlung der arithmetischen Operationen folgendermaßen aus:

```
PROCEDURE Tr(Lkl: SExp): SExp;
       ...
BEGIN
       ...
  x  := HeadSE(Lkl);  (* Operator des Ausdrucks Lkl *)
       ...
  kw := GetSymbolTableEntry(x); (* Suche Operator in Symboltabelle *)
     ...
  IF       kw = KeywordSE[ add ] THEN
    RETURN InBrackets(TrPlus(Lkl))
  ELSIF    kw = KeywordSE[ sub ] THEN
    RETURN InBrackets(TrSub(Lkl))
  ELSIF    kw = KeywordSE[ mul ] THEN
    RETURN InBrackets(TrMul(Lkl))
  ELSIF    kw = KeywordSE[ div ] THEN
    RETURN InBrackets(TrDiv(Lkl))
  ELSIF    kw = KeywordSE[ mod ] THEN
    RETURN InBrackets(TrMod(Lkl))
       ...
END Tr;
```

Abbildung 3.10: Feinstruktur des Compilers



Der Parameter Lkl der Transformationsprozedur repräsentiert den zu verarbeitenden *Scheme*-Ausdruck, der zunächst analysiert wird. In Abhängigkeit vom ermittelten Operator wird dann die Arbeit auf spezielle Einzelfunktionen für die Transformation bestimmter *Scheme*-Konstruktionen verteilt, die die entsprechenden *Modula-2*-Codesequenzen generiert. Eine solche Funktion ist z. B. **TrPlus**:

```
PROCEDURE TrPlus( L: SExp): SExp;
 VAR IntTypeControledL: SExp;
BEGIN
 IntTypeControledL:= IntTypeControl(L) ;
 RETURN ConsSE(HeadSE(TailSE(IntTypeControledL)),
              ConsSE(PlusSEsym,
                     ConsSE(HeadSE(TailSE(TailSE(IntTypeControledL))),
                            CreateSE())))
END TrPlus;
```

Dabei überprüft die Funktion **IntTypeControl**, ob der Typ der zwei Argumente einer binären Funktion numerisch ist. Wenn Typkonflikte auftreten, werden Konvertierungsfunktionen in den Zielcode eingebaut.

Abschließende Ausgabe des *Modula-2*-Codes: Reparser

Die Ausgabe des Translators ist eine baumorientierte Repräsentation (eine Liste) des *Modula-2*-Codes. Sei beispielsweise die *Scheme*-Funktion *fakt* angegeben:

```
(define fakt
  (lambda (n )
     (if (=  n 1)
      1
      (* n   (fakt (- n 1))))))
```

Ihre Übersetzung in die baumorientierte Darstellung des *Modula-2*-Codes sieht folgendermaßen aus:

```
(((PROCEDURE FAKT(N:INTEGER):INTEGER;)BEGIN(IF(((N=1)))THEN RETURN(1)ELSE
 RETURN(((N*(FAKT(((N-1)))))))END;)END FAKT;))
```

Dieser Baum wird danach in die Zieldatei mit Hilfe eines Reparsers ausgegeben. Dabei entsteht folgende *Modula-2*-Prozedur:

```
PROCEDURE FAKT (N: INTEGER):INTEGER ;
BEGIN
  IF (N = 1) THEN
    RETURN 1
  ELSE
    RETURN (N*FAKT((N - 1)))
  END ;
END FAKT ;
```

Der Reparser erweitert das Zielprogramm noch um eine IMPORT-Liste und um Initialisierungsoperationen. In der IMPORT-Liste wird ein Bezug zum Laufzeitsystem hergestellt, in dem eine Reihe vorimplementierter Funktionen zusammengefaßt ist.

3.6 Vergleich mit anderen Compilierungstechniken

Quelltextcompilation hat sich als gängige Technik bei der Implementation funktionaler Sprachen etabliert. Im Abschnitt 1.4.2 hatten wir einen Überblick über die betreffenden Projekte gegeben. Im vorliegenden Abschnitt wollen wir unseren Ansatz einordnen und mit anderen Ansätzen vergleichen. Dabei konzentrieren wir uns insbesondere auf den Vergleich mit dem Scheme-C-Compiler (*SCC*) von Bartlett [Bar93]. Dieser Compiler kommt unserer Übersetzungsstrategie in einer Reihe von grundlegenden Ideen am nächsten.

3.6.1 Vergleich verschiedener Ansätze direkter Übersetzung funktionaler in prozedurale Sprachen

In diesem Abschnitt werden die Grundprinzipien des von uns definierten Compilers angegeben und mit denen anderer verwandter Compiler verglichen. Bei der Implementation einer funktionalen Programmiersprache ist es wichtig, wie die Funktionen repräsentiert werden, wie die Funktionsapplikation durchgeführt wird und wie der Programmablauf gesteuert wird.

In diesem Abschnitt werden die genannten Aspekte des von uns realisierten Compilers mit einigen vorliegenden Compilern, deren zugrunde liegende Techniken als grundlegende Prinzipien der Implementation direkter Transformationen zwischen funktionalen und prozeduralen Sprachen betrachtet werden können, verglichen.

Repräsentation der Funktionen und Realisierung der Funktionsapplikationen

Wenn eine (funktionale) Sprache Funktionen höherer Ordnung unterstützt, ist es üblich, die Funktionen durch Aktivierungsrecords (*activation records*) darzustellen. Ein Aktivierungsrecord kann als ,,Körper" einer Funktion aufgefaßt werden, der im Moment des Aufrufs der Funktion, aktiviert wird.

Funktionsdefinitionen können verschachtelt sein, d. h. eine lokale Funktion F kann innerhalb einer anderen umgebenden Funktion G definiert werden. Alle in der umgebenden Funktion G definierten Variablen sind auch in der lokalen Funktion F gültig. Eine solche Verschachtelung fordert bei jedem Aufruf der lokalen Funktion F auch Zugang zu den in der umgebenden Funktion G definierten Variablen.

Ein Paar, das aus den Adressen der umgebenden Variablen (Umgebung, engl. *einvironment*) und dem Aktivierungsrecord der Funktion F besteht, nennen wir ein Closure.

Bei der Implementation einer Sprache durch ihre Übersetzung in eine andere Sprache bieten sich mehrere Möglichkeiten der Darstellung der Closures bzw. der Aktivierungsrecords an. In [Bar93] werden z. B. *Scheme*-Funktionen in *C*-Funktionen direkt übersetzt. Ein Closure einer Funktion ist ein Record, das aus einem Pointer auf die Funktion und einer für die Funktion geltenden Umgebung besteht.

Bei unserem Compiler haben wir versucht, ein Zielprogramm mit möglichst wenigen abstrakten Datenstrukturen zu erzeugen. Eine in einer prozeduralen Sprache erzeugte Funktion wird in unserem Modell mit Informationen durch einen *Stack* versorgt[9]. Geben wir ein Beispiel, bei dem wir von folgendem Programm ausgehen:

```
letrec
  addfun  = lambda x y . app + x y
  add1    = app addfun 1
in app add1 10
```

Dieses Programm wird bei uns in das *Modula-2*-Programm aus Abbildung 3.11 übersetzt.

Vor dem Aufruf der Funktion *add1* war es notwendig, einen Parameter, mit dem die zweistellige Funktion *addfun* bei der Definition von *add1* partiell parametrisiert wurde, zu speichern. Auf den Wert des Parameters verweisen wir in der Funktion *add1*, die eigentlich die Ergebnisfunktion der partiellen Parametrisierung von *addfun* ist.

[9]Die Funktionsweise des *Stacks* haben wir näher im Abschnitt 3.3 beschrieben. Diese Vorgehensweise ähnelt dem in [PJ92] beschriebenen Ansatz des *Push-Enter* Modells.

Abbildung 3.11: Beispiel eines Zielprogramms in Modula-2

```
MODULE beispiel;
FROM InOut IMPORT
  WriteString, WriteLn, WriteInt;
FROM STACK IMPORT
   Push, Top, MakeNull, Stack, Pop;
VAR
  S      : Stack;
  HlpVar : INTEGER;
  Prog   : INTEGER;

PROCEDURE addfun(x, y: INTEGER):INTEGER;
BEGIN
  RETURN x + y
END addfun;

PROCEDURE add1(a: INTEGER): INTEGER;
BEGIN
  Top(S, HlpVar);
  RETURN addfun(HlpVar, a)
END add1;


BEGIN
  MakeNull(S);
  HlpVar := 1;
  Push(S,HlpVar);
  Prog := add1(10);
  Pop(S);
  WriteString("Result:");
  WriteInt(Prog, 4);
  WriteLn;
END beispiel.
```

Der Programmablauf

Bei der Übersetzung einer funktionalen in eine prozedurale Sprache ist eine wichtige Frage des Compiler-Entwurfs, wie der Programmablauf gesteuert wird. Bei dem in [Bar93] definierten *Scheme*→*C*-Compiler werden *Scheme*-Funktionen in *C*-Funktionen übersetzt. Der Programmablauf der Quellsprache wird also durch transparente Übersetzung in der Zielsprache simuliert (für den Programmablauf wird ein Aufruf-Stack von *C* benutzt). Bartlett's *Scheme*→*C*-Compiler übersetzt also die Funktionsaufrufe direkt.

Eine Optimierung der Funktionsaufrufe besteht z. B. darin, den Code einer Funktion, die nur einmal aufgerufen wird, direkt an der Aufrufstelle zu erzeugen und den Aufruf durch eine *goto*-Anweisung durchzuführen.

Der *ML*-*C*-Compiler [TAL90], der *Scheme*-*C*-Compiler (*Rabbit*-Compiler) [Jr.78] und der *Orbit*-Compiler [KKR$^+$86] (auch ein *Scheme*-*C*-Compiler) basieren auf der Übersetzung des Quellprogramms in *CPS*-Form (*continuation passing style*). Jede Funktion der Quellsprache wird um einen weiteren Parameter erweitert. Das ist die Funktion (*continuation*-Funktion), die anschließend an die Evaluation der z. Z. aktuellen Funktion aufgerufen wird. Darüber hinaus sind alle Funktionsaufrufe innerhalb einer Funktion *Tail-calls*. Die Aufrufe aller Funktionen können als parametrisierte *Goto*-Anweisungen betrachtet werden. Für die Funktionsaufrufe in der Zielsprache

ist ein interessantes Verfahren in [TAL90, Jr.78] eingesetzt: Funktionsaufrufe werden nicht direkt nach *C* übersetzt. Die den Programmablauf führende *apply-like* Funktion[10] ruft zunächst die erste auszuführende Funktion auf. Sie liefert die nächste auszuführende Funktion. Nun wird diese Funktion aufgerufen. Sie liefert wieder die nächste aufzurufende Funktion ... usw. Auf dieser Art und Weise werden in einer Schleife die Funktionen des prozeduralen Zielprogramms verkettet aufgerufen. Dieses Verfahren ermöglicht einen Gewinn an Programmeffizienz. Darüber hinaus wird der *Stack* der Zielsprache nicht belastet (die maximale Tiefe des besetzten *Stacks* ist 2).

Unser Compiler basiert auf einer transparenten Übersetzung der Funktionen, d. h. daß der *Stack* der Zielsprache für die Funktionsaufrufe benutzt wird. Das Übersetzungsverfahren basiert nicht auf *Continuations*.

3.6.2 Vergleich mit Bartletts Transformationstechnik

Der *Scheme→C*-Compiler (*SCC*) von Bartlett [Bar93] kommt unserer Übersetzungsstrategie in seinem Grundansatz am nächsten. Aus diesem Grunde soll hier ein näherer Vergleich beider Quelltextcompiler erfolgen.

Der *Scheme→C*-Compiler von Bartlett ist dem von uns realisierten *Scheme→Modula-2*-Compiler (*SMC*) in folgenden Punkten sehr ähnlich:

- beide Compiler basieren auf einer direkten Übersetzung der Quellsprache in eine prozedurale Sprache (ohne abstrakte Maschinen)
- der Programmablauf beider Compiler wird durch die transparente Übersetzung der Funktionsaufrufe gesteuert (z. B. nicht durch *continuations*).

Der entscheidende Unterschied zwischen beiden Übersetzungsstrategien besteht in der Art des erzeugten Zielcodes. Bei der Bartlettschen Compilationstechnik werden die Typen der funktionalen Quellsprache nicht näher analysiert, so daß in der Zielsprache alle erzeugten Ausdrücke denselben Typ besitzen. Das macht den Übersetzer selbst einfacher, führt aber zur unübersichtlicheren Zielprogrammstrukturen. Die Folge ist zudem eine geringere Effektivität des *SCC*-Compilers im Vergleich zu unserem Ansatz. Wir werden nun einen näheren Vergleich der zwei Transformationsstrategien geben und mit einem Überblick über den *SCC*-Compiler beginnen.

Die grundlegende Übersetzungsstrategie des Scheme-to-C-Compilers *SCC*

Die Übersetzung des *SCC* läuft in folgenden vier Schritten ab:

1. Konvertierung aller Ausdrücke in ihre primitiven Äquivalente, z. B.:

[10] Dieser Mechanismus für die Funktionsaufrufe heißt *UUO handler* [Jr.78].

(or x y) $\Leftrightarrow\!\!\rightarrow$ ((lambda (x thunk) (if x x (thunk))) x (lambda () Y))

2. Transformation in die entsprechende interne Form:

   - Die Argumente der Lambda-Ausdrücke werden in neu generierte kryptische Namen überführt, so daß eine Beziehung zum Originalnamen nicht mehr sichtbar ist (α-*Conversion*). Der Aufruf der o. g. logischen Funktion **or** wird beispielsweise folgendermaßen transformiert:

     ((lambda (x thunk) (if x x (thunk))) x (lambda () Y))

     $\Leftrightarrow\!\!\rightarrow$ ($IF X1006 X1006 ($CALL () T1007))

     Die neuen Namen für die formalen Parameter X und Y der Funktion **or** sind jetzt X1006 und T1007. Der Parameter X1006 wird entsprechend dem Quellprogramm mit der Funktion **and** realisiert und das zweite Argument Y mit der Funktion **thunk**:

```
    ...
(X1006 <- (<APPLY> () L1008
           (X1009 <- X1002)
           (T1010 <- ($LAMBDA L1011 Y1003))
           ($IF X1009 ($CALL () T1010) X1009)))
(T1007 <- ($LAMBDA L1012 Z1004))
    ...
```

     Das Programm hat jetzt die Form einer Deklaration. Funktional kann das folgendermaßen beschrieben werden:

```
LET X1006 = LET X1009 = X1002
              T1010 = LAMBDA L1011 Y1003
            IN IF X1009 THEN T1010 ELSE X1009
    T1007 = LAMBDA L1012 Z1004
IN  ($IF X1006 X1006 ($CALL () T1007))
```

     Dabei wird eine Struktur gebildet, in der nach den Namen der Argumente die gegenwärtigen Angaben gespeichert werden. Bei der Definition der Funktion T1010 durch den λ-Ausdruck LAMBDA L1011 Y1003 sind das z. B. die Variablen L1011 und Y1003.

   - Alle Funktionsaufrufe werden mit einem zusätzlichen Argument erweitert, das ein Hilfsmittel für *tail-calls* ist. Der Aufruf

     $CALL () T1010

bedeutet den Aufruf der Funktion T1010 mit leerem Argument für *tail-calls*.

3. Transformation der Funktionsaufrufe und optimierende Transformation

   - Funktionsaufrufe, bei denen die Funktionen λ-Ausdrücke sind, werden jetzt in eine solche Form transformiert, die lambda-Bindungen enthält.[11]

```
(<APPLY> () L1008
  (X1009 <- X1002)
  (T1010 <- ($LAMBDA L1011 Y1003))
  ($IF X1009 ($CALL () T1010) X1009)))

===>   ($CALL ()
         ($LAMBDA L1008
            ($IF X1009 ($CALL () T1010) X1009)
                    X1002
                   ($LAMBDA L1011 Y1003))
```

   Symbolisch beschrieben:

```
LET X1009 = X1002
    T1010 = LAMBDA L1011 Y1003
IN  IF X1009 THEN ($CALL () T1010)
              ELSE X1009

===>   (LAMBDA L1008 IF X1009 THEN  ($CALL () T1010)  ELSE X1009)
              X1002
              (LAMBDA L1011 Y1003)
```

   - Optimierende Transformation mit *Rewriting Rules*[12]
     Jede Variable in einer *Apply*-Struktur hat einen ihr entsprechenden Ausdruck bzw. eine andere Variable (Adresse). Bei der Übersetzung werden die Referenzen benutzt und direkt in den Ausdrücken abgelegt. Die Codesequenz

     ($IF X1006 X1006 ($CALL () T1007))) (*)

     ist mit folgender Umgebung verbunden

```
(X1006 <- (<APPLY> () L1008
```

[11] Diese Transformation kann als eine „*let* to *lambda*" - Transformation definiert werden.

[12] Diese Transformation mit dem Ziel der Optimierung des Zielodes wurde bei dem *Scheme-to-C* Compiler von Kranz verwendet [KKR+86]. Dieser Compiler ist der Effizienz nach mit den damaligen effizientensten Compilern prozeduraler Sprachen verglichen worden.

```
(X1009 <- X1002)
(T1010 <- ($LAMBDA L1011 Y1003))
($IF X1009 ($CALL () T1010) X1009)))
```

X1006 ist also durch eine „let"-Struktur definiert, deren Rumpf wieder eine *if*-Struktur ist. Der logische Ausdruck dieser *if*-Sruktur ist die Variable X1009, deren Wert X1002 ist. Wenn X1002 der logische Wert *true* ist, wird der durch den λ-Ausdruck Y1003 definierte Ausdruck T1010 (Funktionsaufruf) geliefert. Der ist aber ein Parameter für die *if*-Struktur auf der höheren Ebene. Falls der Parameter *true* ist, wird die Konstante C1014 geliefert usw ... Die Codesequenz

($IF Z1004 C1014 C1013)

wiederholt sich. Deshalb wird hier eine neue Prozedur eingeführt.

4. C-Code-Generierung
   Aufgrund einer Analyse des Quellcodes wird entschieden, ob Scheme-Funktionen als C-Prozeduren übersetzt werden und wie das Zielprogramm in Codesequenzen geteilt wird.

Die neueren Versionen des *SCC*-Compilers erzeugen einen etwas geänderten C-Code. Statt Funktionen, werden C-Prozeduren erzeugt und die *return*-Effekte sind mit Hilfe eines *Stack* erzielt worden. Obiges Beispiel wird dabei wie folgt übersetzt:

```
TSCP  test_sample(x2002, y2003, z2004)
        TSCP  x2002, y2003, z2004;
{
        PUSHSTACKTRACE(t2025);
        if (FALSE(x2002))  goto L2032;
        if (FALSE(y2003))  goto L2032;
        POPSTACKTRACE(_TSCP(0));
  L2032:
        if (TRUE(z2004))  goto L2033;
        POPSTACKTRACE(_TSCP(4));
  L2033:
        POPSTACKTRACE(_TSCP(0));
}
```

In der Abbildung 3.12 ist ein vollständiges Beispiel der Übersetzung angegeben.

Nach diesem kurzen Abriß der Transformationsstrategie des *SCC* kommen wir jetzt zu einem näheren Vergleich mit dem von uns implementierten *Scheme-Modula-2*-Compiler (*SMC*). Wir gehen dabei schrittweise auf einzelne Aspekte ein.

Abbildung 3.12: Übersetzungsschritte des *SCC*

```
(define (sample x y z)
  (if (or (and (x y ) z)
    0
    1 ))
```

$\overset{1}{\Longrightarrow}$

```
(define (sample x y z)
  (if
   ((lambda (x thunk)
       (if x x (thunk)))
     ((lambda (x thunk)
       (if x (thunk) x))
          x
          (lambda () Y))
    (lambda () Z))
   0
   1))
```

$\overset{2}{\Longrightarrow}$

```
(DEFINE S1000
  ($LAMBDA L1001
   ($IF
     (<APPLY> () L1005
      (X1006
        <-
         (<APPLY> () L1008
           (X1009 <- X1002)
           (T1010 <- ($LAMBDA L1011 Y1003))
           ($IF X1009 ($CALL () T1010) X1009)))
      (T1007 <- ($LAMBDA L1012 Z1004)
      ($IF X1006 X1006 ($CALL () T1007)))
     C1014
     C1013)))
```

$\overset{3}{\Longrightarrow}$

```
($DEFINE S1000
  ($LAMBDA L1001
      (<APPLY> L1001 L1020
       (Y1022 <- ($LAMBDA L1024 ($IF Z1004 C1014 C1013)))
       ($IF X1002
            ($IF Y1003 C1014  ($CALL L1001 Y1022))
            ($CALL L1001 Y1022)))))
```

$\overset{4}{\Longrightarrow}$

```
TSCP  test_sample(x1002, y1003, z1004)
        TSCP  x1002, y1003, z1004;
{
        if  (FALSE(x1002))  goto L1032;
        if  (FALSE(y1003))  goto L1032;
        return(_TSCP(0));
L1032:
        if  (TRUE(z1004))  goto L1033;
        return(_TSCP(4));
L1033:
        return(_TSCP(0));
}
```

Erzeugung von Funktionen der Zielsprache im *SCC* und im *SMC*

Alle globalen Funktionen von *Scheme* werden durch den *SCC* in C-Funktionen übersetzt. Die *Scheme*-Funktionen werden in der Übersetzung auf kleine Unterprozeduren aufgeteilt, wie z. B. die Prozedur L1024 im vorherigen Beispiel. Danach wird entschieden, wo diese Codesequenz abgelegt wird. Diese Unterteilung des Programms macht das erzeugte Programm schwer verständlich.

In dem von uns realisierten *Scheme-Modula-2*-Compiler *SMC* werden alle *Scheme*-Funktionen in *Modula-2*-Prozeduren übersetzt. Der Rumpf der Funktion wird jedoch nicht unterteilt, sondern jeder Ausdruck wird in einen Ausdruck von *Modula-2* übersetzt, und zwar in der Reihenfolge des Quellprogramms. Möglich sind eventuelle Kontextergänzungen, die wir bei den Transformationsregeln in Abschnitt 3.3.1 angegeben haben.

Der zur Illustration des *SCC* verwendete *Scheme*-Ausdruck

```
(define (sample x y z)
  (if (or (and (x y ) z)
    0
    1))
```

wird mit dem *SMC* wie folgt nach *Modula-2* übersetzt:

```
PROCEDURE SAMPLE (X: BOOLEAN;
                  Y: BOOLEAN;
                  Z: BOOLEAN):INTEGER;
BEGIN
  IF((X AND Y) OR Z) THEN
    RETURN 0
  ELSE
    RETURN 1
  END;
END SAMPLE;
```

Die elementaren Programmabschnitte des mit dem *SCC* erzeugten *C*-Programms werden mit Hilfe von *Goto*-Befehlen verbunden, was einen negativen Einfluß auf die Lesbarkeit und die Beziehung zum Originalprogramm hat.

Übersetzung von Standardfunktionen

Im obigen Beispiel wurde gezeigt, wie der *SCC* die im funktionalen Programm enthaltene Menge der Funktionen erweitert, indem Standardfunktionen in λ-Ausdrücke überführt werden. Für „or" ergibt sich beispielsweise:

(or x y) $\Leftrightarrow\!\!\rightarrow$ ((lambda (x thunk) (if x x (thunk))) x (lambda () Y))

Nach diesem Übersetzungsschritt wird die Standardfunktion genauso wie benutzerdefinierte Funktionen behandelt. Das macht den Zielcode schlechter lesbar und größer. Die Beziehung zum Originalprogramm wird verloren, obwohl es eine entsprechende Standardfunktion in *C* gibt. Schließlich kann es auf diese Weise Effektivitätsverluste geben.

In unserem Compiler *SMC* versuchen wir, Standardfunktionen der Zielsprache *Modula-2*, wenn möglich, zu verwenden.

Bezeichner

Der *SCC*-Compiler behält die Originalbezeichner nicht bei. Die Beziehung zum Originalprogramm wird damit geringer erkennbar. Im *SMC* werden Originalbezeichner beibehalten. Nur wenn Kollisionen auftreten, werden Bezeichner verändert.

if-Kontrollstruktur

Die Kontrollstruktur *if* wird durch den *SCC* in eine *if*-Anweisung des *C*-Programms übersetzt. Durch die Unterteilung des Programms auf die Kodesequenzen kommen im erzeugten C-Programm zusätzliche *if*-Konstrukte hinzu.

Mit dem Verlust der Originalbezeichner verliert die übersetzte *if*-Struktur ihre Beziehung zur ihr entsprechenden *if*-Konstruktion des Quellprogramms. Als Beispiel nehmen wir das folgende *Scheme*-Programm:

```
(define if1
   (lambda (n)
      (if (< n 10)
         (quote kleiner)
         (quote groesser))))
```

Die mit dem *SCC* erzeugte Übersetzung dieses Programms ist:

```
TSCP  if1_if1(n2006)
        TSCP  n2006;
{
        PUSHSTACKTRACE(t2023);
        if  (BITAND(BITOR(_S2CINT(n2006),
                          _S2CINT(_TSCP(40))),
                    3))  goto L2025;
/* ! */ if  (LT(_S2CINT(n2006), _S2CINT(_TSCP(40))))
             goto L2027;
        POPSTACKTRACE(c2014);
```

```
L2027:
        POPSTACKTRACE(c2015);
L2025:
/* ! */ if  (TRUE(scrt2__3c_2dtwo(n2006, _TSCP(40))))
             goto L2029;
        POPSTACKTRACE(c2014);
L2029:
        POPSTACKTRACE(c2015);
}
```

Der *SCC*-Compiler hat zwei *if*-Strukturen erzeugt. Die Strukturen sind mit /* ! */ gekennzeichnet.

Die Übersetzung des *SMC* lautet:

```
PROCEDURE IF1(N: INTEGER): SExp;
BEGIN
  IF (N<10) THEN
    RETURN QuoteSE('KLEINER')
  ELSE
    RETURN QuoteSE('GROESSER')
  END;
END IF1;
```

In diesem Code gibt es nur eine *if*-Kontrollstruktur, die der *if*-Struktur des Quellcodes entspricht.

do-Kontrollstruktur

Die *do*-Kontrollstruktur ist eine iterative Sprachkonstruktion. Dieses Sprachelement kann im erzeugten prozeduralen Code mit Hilfe eines *goto*-Befehls implementiert werden. Eine bessere Übersetzung wäre mit Hilfe der *do*-Anweisung in *C* möglich. Geben wir hier ein Beispiel aus [Bar93] an:

```
(define (print_1_to_limit limit)
  (do ((i 1 (+ i 1)))
      ((> i limit) 'done)
      (display i)))
```

Die Übersetzung des *SCC* ist:

```
TSCP  do1_print__1__to__limit(l2006)
        TSCP  l2006;
{
        TSCP  X1;

        PUSHSTACKTRACE(t2034);
        X1 = _TSCP(4);
L2037:
        if (BITAND(BITOR(_S2CINT(X1),
                         _S2CINT(l2006)),
                   3))  goto L2039;
        if (LTE(_S2CINT(X1), _S2CINT(l2006)))  goto L2043;
        POPSTACKTRACE(c2025);
L2039:
        if (FALSE(scrt2__3e_2dtwo(X1, l2006)))  goto L2043;
        POPSTACKTRACE(c2025);
L2043:
        scrt6_display(X1, EMPTYLIST);
        if (BITAND(BITOR(_S2CINT(X1),
                         _S2CINT(_TSCP(4))),
                   3))  goto L2047;
        X1 = _TSCP(IPLUS(_S2CINT(X1),
                         _S2CINT(_TSCP(4))));
        GOBACK(L2037);
L2047:
        X1 = scrt2__2b_2dtwo(X1, _TSCP(4));
        GOBACK(L2037);
}
```

Der von unserem *SMC*-Compiler erzeugte *Modula-2*-Code sieht folgendermaßen aus:

```
PROCEDURE print_1_to_limit(limit: CARDINAL):SExp;
VAR
  i: CARDINAL;
  retName: SExp;
BEGIN
  i := 1;
  REPEAT
    WriteCard(i, 4);
    i := i+1;
  UNTIL (i > limit);
  retName := QuoteSE("DONE");
```

```
  RETURN (retName)
END print_1_to_limit;
```

Typisierung

Die mit dem *SCC* aus der Sprache *Scheme* erzeugten C-Ausdrücke haben alle denselben Typ (TSCP). Zum Beispiel hatte die Funktion *sample* (Abbildung 3.12) den numerischen Typ und die formalen Parameter den logischen Typ. Um (im Beispiel) logische Operationen über den Argumenten der Funktionen durchzuführen, werden die in C entwickelten speziellen Funktionen aufgerufen (z. B. TRUE, FALSE).

Mit unserem Verfahren haben wir beim *SMC* versucht, den funktionalen Ausdruck in ein *Modula-2*-Programm zu übersetzen, so daß möglichst viele Variablen elementare Standardtypen besitzen. Wenn eine Variable keinen einfachen Typ besitzen kann, wird der Listentyp genommen.

Zusammenfassendes Beispiel

Abschließend soll ein Beispiel (Fakultätsfunktion) angegeben werden, bei dem der Typ der Funktion und der Typ des Arguments numerische Typen sind. Für den ursprünglichen *Scheme*-Ausdruck geben wir für die beiden Compiler *SCC* bzw. *SMC* die entsprechend erzeugten *C*- bzw. *Modula-2*-Programme an.

Scheme-Ausdruck:

```
(define fakt
  (lambda (n)
    (if  (= n 1)
       1
       (* n (fakt (- n 1))))))
```

SCC:

```
TSCP  fakt_fakt(n2006)
        TSCP  n2006;
{
        TSCP  X2, X1;

        PUSHSTACKTRACE(t2036);
        if (BITAND(BITOR(_S2CINT(n2006),
                         _S2CINT(_TSCP(4))),
                   3))  goto L2039;
        if (NEQ(_S2CUINT(n2006),
```

```
                _S2CUINT(_TSCP(4))))  goto L2043;
        POPSTACKTRACE(_TSCP(4));
L2039:
        if (FALSE(scrt2__3d_2dtwo(n2006, _TSCP(4))))  goto L2043;
        POPSTACKTRACE(_TSCP(4));
L2043:
        if (BITAND(BITOR(_S2CINT(n2006),
                         _S2CINT(_TSCP(4))),
                   3))  goto L2046;
        X2 = _TSCP(IDIFFERENCE(_S2CINT(n2006),
                               _S2CINT(_TSCP(4))));
        goto L2047;
L2046:
        X2 = scrt2__2d_2dtwo(n2006, _TSCP(4));
L2047:
        X1 = fakt_fakt(X2);
        if  (BITAND(BITOR(_S2CINT(n2006),
                          _S2CINT(X1)),
                    3))  goto L2049;
        POPSTACKTRACE(_TSCP(ITIMES(FIXED_C(n2006),
                                   _S2CINT(X1))));
L2049:
        POPSTACKTRACE(scrt2__2a_2dtwo(n2006, X1));
}
```

SMC:

```
PROCEDURE FAKT(N: INTEGER): INTEGER;
BEGIN
  IF (N = 1) THEN
    RETURN 1
  ELSE
    RETURN (N * FAKT((N - 1)))
  END;
END FAKT;
```

Das *Modula-2*-Zielprogramm ist rein-numerisch und gut verständlich. Konvertierungsfunktionen werden nicht aufgerufen. Es gibt keine redundanten Aktionen im Zielcode. Der *SCC*-Zielcode ist unverständlich, hat keine rein-numerischen Typen und ist unstrukturiert.

Laufzeitvergleich

Wir haben die zwei Compiler am Beispiel von sechs Testprogrammen verglichen:

- *diff*: Differenzierung eines Ausdrucks mit arithmetischen Operationen
- *fib*: Berechnung der n-ten Fibonacci-Zahlen
- *fakt*: Fakultätsfunktion
- *queen*: Lösung des Problems der n-Damen
- *tak*: Takeiushi-Funktion
- *union*: Berechnung der Vereinigungsmenge von Mengen, die als Elemente einer Liste repräsentiert werden.

In der Tabelle in der Abbildung 3.13 sind die Ergebnisse dieses Vergleichs zusammengefaßt.

Abbildung 3.13: Laufzeitvergleich der *SCC*- und *SMC*- Compiler

	SCC [ms]	*SMC [ms]*	Vergleich
(diff (plus (mal x x) 2))	0,21	0,061	-70,95%
(fib 30)	3.830,00	2.280,00	-40,47%
(fakt 10)	0,115	0,067	-41,13%
(queen 8)	42,62	40,97	-3,89%
(tak 11 7 50)	2.200,00	1.740,00	-20,91%
union [13]	1,105	1,018	-7,92%

Der Vergleich gibt den Laufzeitgewinn des *SMC* im Vergleich zum *SCC* an. Für (*fib* 30) ergibt sich beispielsweise eine Effektivitätssteigerung um 40,47%. In allen Fällen ist der von uns implementierte *SMC* schneller als der *SCC*. Ein hoher Effektivitätsgewinn ist insbesondere bei den Programmen zu verzeichnen, die in unserem Modell „rein" typisiert werden können, wie z. B. *fakt*, *fib*, *diff* und *plus*.

[13] Das Programm wurde mit drei Parametern aufgerufen: eine Liste mit 22 Elementen und zwei mit 2 Elementen.

Kapitel 4

Stack-basierte Implementation und Quelltexttransformation

Im vorherigen Kapitel ist eine direkte Übersetzung einer funktionalen Sprache in eine prozedurale Sprache angegeben worden. In diesem Kapitel analysieren wir in wesentlichen Punkten, wie die im Abschnitt 2.3 beschriebene Stack-basierte Implementationstechnik ein Zielprogramm in einer prozeduralen Sprache erzeugen kann.

Die Natur der abstrakten *SECD*-Maschine ist prozedural. Deshalb gibt es keine größeren Schwierigkeiten der Realisierung eines Compilers, dessen Back-end die Sprache der *SECD*-Maschine in eine höhere prozedurale Programmiersprache übersetzt.

In den folgenden Abschnitten geben wir die grundlegenden Prinzipien eines auf dieser Technik basierenden Compilers an.

Aufgrund der definierten Semantik und der Komponenten der Maschine ist es nicht schwer, das *Laufzeitsystem* des Compilers festzulegen. Das Laufzeitsystem des Compilers besteht aus folgenden Komponenten:

- Vier Komponenten der Maschine (S E C und D)
- Prozeduren, die den Maschinenbefehlen entsprechen.

Nun wird das Prozeß der Evaluation in einem prozeduralen Modell (d. h. die Übersetzung in eine der prozeduralen Sprachen) definiert. Bei der Evaluation des *SECD*-Codes bieten sich zwei Möglichkeiten an, die wir in den Grundzügen charakterisieren wollen.

4.1 Möglichkeit I: Simulation der Transitionen

Die erste Möglichkeit der Evaluation ist ganz natürlich: Das *SECD*-Programm wird durch eine Zustandmanipulation ausgeführt. Das Zielprogramm besteht in diesem

Fall aus folgenden Teilen:

- Initialisierung des Listenmechanismus und der Maschinenkomponenten (im Register *C* wird der gesamte *SECD*-Code abgelegt)
- Evaluation des Codes bis zum Vorkommen des Befehls *STOP*. Die Evaluation wird entsprechend der operationellen Semantik (*Transitionen*) der *SECD*-Maschine durchgeführt.
- Ausdrucken des ersten Elements des Stacks (das Register *S*), was das Ergebnis der Evaluation des Programms ist.

In Pseudocode würde das Prozess folgendermaßen aussehen:

```
begin programm;
  Init;
  while (not (first(C)=STOP)) do
    Execute_One_Operation
  end_while;
  Print(first_el(S))
end programm.
```

Execute_One_Operation nimmt das erste Element vom Register *C* und ruft die dem Machinenbefehl entsprechende Prozedur auf. Das ist eine Lösung, die bei allen Implementationstechniken möglich ist.

4.2 Möglichkeit II: Implementation ohne *C*-Register

Die Grundidee besteht darin, aus dem Maschinencode, der sich im *C*-Register befindet, ein prozedurales Programm zu erzeugen. Das neue Laufzeitsystem (der prozeduralen Sprache) besteht nun aus drei Komponenten: *S*, *E*, und *D*. Die Komponente *C* ist das Programm selbst. Jeder Maschinenbefehl hat in der prozeduralen Sprache eine ihm entsprechende Prozedur. Der Aufruf des Maschinenbefehls wird in den Aufruf der Prozedur übersetzt. Die Prozeduren behandeln die Laufzeitkomponenten genauso wie die ihnen ensprechenden Maschinenbefehle (nach den im Abschnitt 2.3 angegebenen Transitionen). Die Übersetzung aller Maschinenbefehle, die auch das Register *C* ändern, muß genauer definiert werden.

Die einzige Schwierigkeit bei der Implementation besteht darin, daß das *C*-Register während des Programmablaufs geändert wird. Dadurch kann man nicht immer aus dem Code des *C*-Registers eine feste Befehlsfolge generieren. Vom folgendem *SECD*-Programm ist das z. B. möglich:

(LD (0.0) CAR LDC 1 ADD)

Eine Lösung im prozeduralen Pseudocode wäre:

```
begin programm;
   ...
  ld(0, 0);
  car;
  ldc(1);
  add;
end programm;
```

Die Prozeduren *ld*, *car*, *ldc* und add gehören zum Laufzeitsystem und entsprechen den gleichnamigen *SECD*-Maschinenbefehlen. Als nächstes geben wir ein *SECD*-Programm an, das in eine prozedurale Sprache auf diese Art und Weise nicht übersetzt werden kann:

(LDF (LD (0.0) LD (1.1) CONS RTN))

Die Transition für LDF lautet:

$s\ e\ (LDF\ c'.c)\ d \rightarrow ((c'.e).s)\ e\ c\ d$

Das bedeutet, daß der Code einer (mit einem λ-Ausdruck definierten) Funktion auf den *Stack* (*S*-Register) gelegt wird, um später aufgerufen werden zu können. Die für diese Übersetzungsstrategie ,,problematischen" Befehle sind:

LDF, AP, RTN, RAP und *SEL*

Andere Befehle können in die ihnen entsprechenden Aufrufe der Prozeduren des Laufzeitsystems übersetzt werden. Geben wir nun für jeden ,,problematischen" Maschinenbefehl eine Möglichkeit seiner Realisierung in der prozeduralen Zielsprache an.

LDF

Der *SECD*-Code jeder lokalen Funktion kann z. B. in einer für alle lokalen Funktionen gemeinsamen Prozedur (sei sie *LokProz* genannt) der prozeduralen Sprache abgelegt werden. Auf das Register *S* kann nur der Bezeichner der Funktion gelegt werden. Beim Aufruf der Funktion wird die Prozedur *LokProz* aufgerufen. Die Prozedur *LokProz* hat nur einen Parameter, der eine Bezeichnung einer lokalen Funktion ist. In der Funktion *LokProz* wird nur die Codesequenz, die der Bezeichnung entspricht, evaluiert. Die Funktion *LokProz* sieht folgendermaßen aus:

```
PROCEDURE LokProz(Bezeichner: UnivTyp);
BEGIN
  CASE Konvert_to_Int(Bezeichner) OF
     1 : Code_von_funktion_1|
     2 : Code_von_funktion_2|
       ...
     n : Code_von_funktion_n
  END
END LokProz;
```

AP

Das ist der Aufruf einer Funktion, deren Code sich schon in der Prozedur *LokProz* befindet. Die Transition für *AP* lautet:

$$((c'.e')\ v.s)\ e\ (AP.c)\ \mathrm{d} \rightarrow NIL\ (v.e')\ c'\ (s\ e\ c.d)$$

Am Anfang des Stacks befindet sich jetzt der Bezeichner der Funktion, die aufgerufen wird. Der Aufruf endet mit dem Aufruf der Funktion *LokProz* mit dem Argument, das oben auf dem Stack liegt. Das *Environment* und der *Dump* (Registern E und D) werden der Transition entsprechend geändert.

RTN

RTN ist für die Programmevaluation nicht mehr nötig, weil die *return*-Effekte mit Hilfe von *returns* in der prozeduralen Sprache erreicht werden. Der *Stack* und das *Environment* werden nach der *Transition* von *RTN* geändert.

RAP

RAP ist eine rekursive Variante des *AP*-Befehls. Die Transition von *RAP* lautet:

$$((c'.e')\ \mathrm{v.s})\ (\Omega.e)\ (RAP.\mathrm{c}) \rightarrow NIL\ rplaca(e',\ v)\ c'\ (s\ e\ c.d)$$

Der einzige Unterschied besteht in der Behandlung des Environments. Der prozedurale Code wird wie beim *AP-Befehl* erzeugt und das *Environment* wird entsprechend der Transition für RAP behandelt.

SEL

Der *SEL* Befehl entspricht dem *if*-Befehl einer prozeduralen Sprache. Die *Transition* von *SEL* lautet:

$$x.s \text{ e } (SEL\ c_t\ c_f.c\ )\ \text{d} \rightarrow s\ e\ c_x\ (c.d)$$

c_x ist dabei c_t bzw. c_f, je nach dem, ob der *SEL*-Befehl auf dem Anfang des *Stacks* das logische *TRUE* oder *FALSE* findet.

Der vollständige bedingte Ausdruck, dessen Konstruktion mit *SEL* anfängt, sieht folgendermaßen aus:

$$(\ ...\ SEL\ (c_t\ JOIN)\ (c_f\ JOIN)\ c)$$

Im Pseudocode hat die Übersetzung des Ausdrucks folgende Struktur:

```
...
if konvert_to_bool(stack_top) then
    Codesequenz_von_c_t
else
    Codesequenz_von_c_f
endif;
Codesequenz_von_c;
...
```

Beispiel:

Übersetzen wir den folgenden funktionalen Ausdruck:

```
((lambda (x y)
   (add y  (if (leq x y) x (quote 1)))
 )
 2 3)
```

Die Übersetzung des Programms führt im ersten Schritt zu folgendem *SECD*-Code:

```
(LDC NIL) | LDC 3 | (CONS) | LDC 2 | CONS |
(LDF (LD(0.1) LD(0.0) LD(0.1) LEQ  <<-- Lambda-Funktion
SEL (LD (0.0) JOIN) (LDC 1 JOIN)  ADD  (RTN))) | (AP) | (STOP)
    ~~~~1~~~~~~~~~  ~~~~~2~~~~~   ~3~
```

Die anschließende Übersetzung in den prozeduralen Code (in *Modula-2*) lautet:

```
MODULE Programm;
 ...
PROCEDURE LokProz(Bezeichner: UnivType);
BEGIN
  CASE ConvertToInt(Bezeichner) OF
```

```
      1 : LD(0,1); LD(0,0); LD(0,1); LEQ;
          IF (ConvertToBool (STop()) THEN
            LD(0.0)   (* ~~~1~~~ *)
          ELSE
            LDCint(1) (* ~~~2~~~ *)
          END
          ADD;       (* ~~~3~~~ *)
          RTN; |
    END
  END
   ...
  PROCEDURE AP;
  BEGIN
   ...
  LokProz(StackTop());
   ...
  END AP;
   ...
  BEGIN
   ...
    LDCt(NIL);
    LDCint(3);
    CONS;
    LDCint(2);
    CONS;
    LoadFun(FunBezeich(), CurEnvironment());
    AP;
   ...
  END Programm.
```

Die Prozedur *FunBezeich* gibt eine natürliche Zahl zurück, der als Bezeichner der neuen Funktion verwendet wird. Die Prozedur CurEnvironment() liefert die z. Z. aktuelle Umgebung (Environment - *E*-Register). STop() gibt das erste Element des Stacks zurück und nimmt es vom Stack weg. Die o. g. Hilfsfunktionen ändern entsprechend den Transitionen die Maschinenkomponenten.

In diesem Kapitel haben wir gezeigt, wie ein *SECD*-basierter Compiler so implementiert werden kann, daß Code einer prozeduralen Sprache erzeugt wird. Zwei Möglichkeit wurden analysiert:

- Simulation der Transitionen
- Erzeugung des prozeduralen Zielprogramms aus dem Code der abstrakten Maschine.

Die zweite Möglichkeit ist dabei effizienter. Weitere Optimierungen des Programmablaufs wären z. B. Typisierung des Zielprogramms, Evaluation ohne Registern D und E usw.

Der Compilationsprozeß könnte aufgrund dessen, daß der Zielcode in einer prozeduralen Sprache erzeugt wird, einfacher realisiert werden. Ein gutes Beispiel dafür ist die Übersetzung einer *if*-Struktur einer funktionalen Sprache, die gleich in eine *if*-Konstruktion einer prozeduralen Sprache übersetzt werden kann.

Kapitel 5

Graph-Reduktion-basierte Implementationstechnik und Quelltexttransformation

In diesem Kapitel führen wir eine Möglichkeit der Erzeugung prozeduralen Codes bei einer auf Graph-Reduktion basierenden Implementationstechnik einer funktionalen Sprache ein.

Als Beispiel einer auf der Graph-Reduktion basierenden Implementationstechnik wird das im Abschnitt 2.4 dargestellte Implementationsverfahren durch die SK-Kombinatoren zugrunde gelegt.

Zunächst wird die Technik auf dem Implementationsniveau dargestellt und die „reine" Graph-Reduktion mit der Quelltexttransformation in Verbindung gebracht. Danach wird eine mögliche Modifikation des Implementationsverfahrens angegeben, die sich aus der Erzeugung des Zielprogramms in einer höheren prozeduralen Programmiersprache ergibt. Die hier vorgeschlagene Strategie wurde von uns in einer Implementation umgesetzt.

5.1 Problemstellung

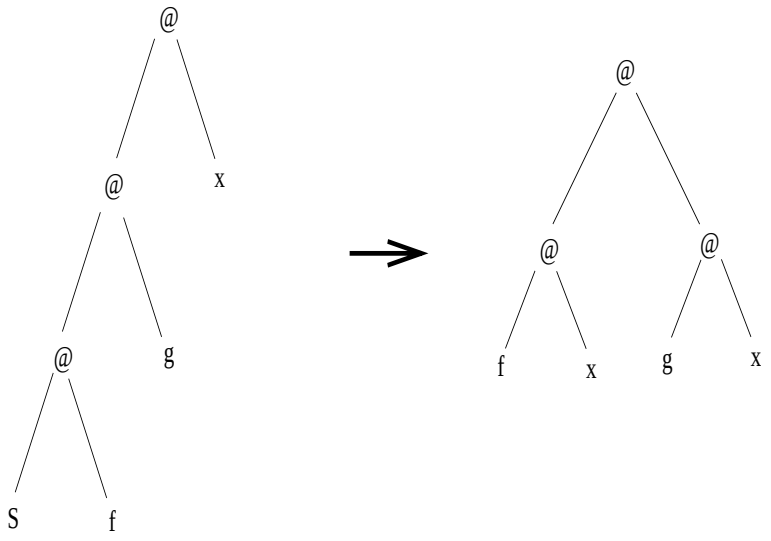
Der Programmablauf prozeduraler Sprachen wird durch eine sequentielle Ausführung von Anweisungen definiert. Das Programm hat dabei eine feste Form, d. h. die Struktur des Programms wird nicht geändert.

Bei der Graph-Reduktion besteht das Wesen der Evaluation in der Umformung des Graphen, der das funktionale Programm repräsentiert. Nehmen wir die Reduktionsregel des S-Kombinators als Beispiel:

$$S\ \mathrm{f}\ \mathrm{g}\ \mathrm{x} \rightarrow \mathrm{f}\ \mathrm{x}\ (\mathrm{g}\ \mathrm{x})$$

In der Abbildung 5.1 ist die baumorientierte Repräsentation dieser Reduktion dargestellt.

Abbildung 5.1: S-Reduktion



Der nächste Befehl wird erst nach der Umwandlung des Graphen bekannt, d. h. daraus kann kein prozedurales Programm im Sinne der sequentiellen Abarbeitung der prozeduralen Befehle, die den Reduktionsschritten entsprechen, erzeugt werden. So besteht das Problem der Erzeugung eines prozeduralen Zielcodes darin, den kombinatorischen Graphen möglichst effektiv im prozeduralen Modell auszuwerten. Wir werden in den folgenden Abschnitten das Laufzeitsystem und die Arbeitsweise des SK-Programmevaluators angeben.

5.2 Laufzeitsystem und Schritte der Programmevaluation

Die Übersetzung einer funktionalen Sprache in den kombinatorischen Term analysieren wir an dieser Stelle nicht ausführlicher. Die Grundideen wurden im Abschnitt 2.4 genannt. Mehr darüber kann man z. B. in [Dil88] [Jon87] [Rev88] und [FH88] finden.

Der kombinatorische Term wird mit Hilfe der Evaluationskomponenten des Laufzeitsystems bis zur *WHNF*-Form reduziert. Die für uns wichtigen Komponenten des

Laufzeitsystems sind:

- der zu reduzierende kombinatorische Term
- Stack (*spine*)
- Prozeduren für die Reduktion des Graphen.

Die ganze Evaluation des SK-Graphen kann in zwei grundlegende Schritte unterteilt werden:

1. Rekonstruktion des Stacks
2. Reduktion.

Wenn der Graph in der *WHNF*-Form vorliegt, bedeutet das, daß die Evaluation beendet ist. In der folgenden Abbildung ist der *Modula-2*-Code der Hauptfunktion für die Evaluation des Graphen angegeben [MB92a]:

```
PROCEDURE Eval (VAR Graph: SExp): SExp;
  VAR c: SExp;
BEGIN
  IF NotEnoughStorageSE() THEN
     CheckMemory   (* Falls es keinen ausreichenden Speicher gibt,*)
                   (* wird der garbage collector aufgerufen *)

  END;
  INC(STop);
(* STop ist der Anfang des stacks. *)

  Spine[STop].no := SBase;
  SBase := STop;
(* Fuer jede Reduktion soll ein neuer Stackanfang*)
(* definiert und der alte gespeichert werden. *)

  UnWind(Graph);
(* Rekonstruktion des Stacks *)

  c := Spine[STop].ex;
(* c ist der Kombinator auf dem Stackanfang *)

  inWHNF := FALSE;
(* inWHNF := TRUE, wenn der z.Z. analysierte Ausdruck nicht reduzierbar ist *)

  WHILE IsTypeSE(c, Combinator) AND
                (GetSymbolTableEntry(c) <> KeywordSE[ cons ]) AND
                NOT inWHNF DO
(* Solange c ein Kombinator und *)
(* kein Listenkonstruktor ist und *)
```

```
(* der z.Z. aktuelle Ausdruck reduzierbar ist. *)

    OneRed;
(* Durchfuehrung der dem Kombinator c *)
(* entsprechenden Reduktion mit *)
(* der Rekonstruktion des Stacks (mit UnWind).*)

    c := Spine[STop].ex;
(* c ist wieder der Kombinator am Anfang des Stacks. *)

  END;

  inWHNF := FALSE;
  STop := SBase;
  SBase := Spine[STop].no;
  DEC(STop);
(* Wiederherstellung des vor der Reduktion des Graphen aktuellen *)
(* Zustands des Stacks. *)

  RETURN Graph;

(* Der nicht mehr reduzierbare Graph ist das Ergebnis der Evaluation*)

END Eval;
```

Auf der Grundlage des eingeführten SK-Evaluationssystems werden wir im folgenden einen Compiler angeben, der eine funktionale in eine prozedurale Sprache überführt.

5.3 Quelltexttransformation durch Graph-Reduktion

Im Abschnitt 5.1 haben wir begründet, daß die Transformation des funktionalen Programms in einen SK-Term für die weitere Übersetzung in eine prozedurale Sprache nicht geeignet ist. Eine Lösung für die Erzeugung prozeduralen Codes besteht darin, den kombinatorischen Term ins Zielprogramm abzulegen und die Prozedur des Laufzeitsystems für die Reduktion des SK-Terms aufzurufen. Das Zielprogramm würde in diesem Fall aus folgenden Teilen bestehen:

- Initialisierung des Systems
- Erzeugung der graphorientierten Darstellung des kombinatorischen Terms
- Aufruf der Funktion für die Evaluation des Graphen (im o.g. Quelltext ist das die Funktion *Eval*)

- Die Rückgabe des Resultats der Auswertung.

Ein SK-Term wird im Rechner auf die gleiche Art und Weise wie Listen dargestellt. So wird der in der Abbildung 2.5 auf Seite 40 dargestellte kombinatorische Term folgendermaßen bezeichnet:

$$((\ldots (Komb.Arg_1).Arg_2 \ \ldots \ ).Arg_n)$$

Zeigen wir nun, wie die Fakultätsfunktion *fak*

```
letrec
    fak = lambda n. if (= n 1) then 1
                                else (fak (- n 1))
in (fak 7)
```

aufgrund des oben definierten Verfahrens übersetzt wird.

Beispiel:

```
...

BEGIN
  InitSE;
  FN := ConsSE(ConsSE(
                    ConsSE( QcSE("C"),
                            QcSE("I")),
                    QIntSE(7)),
             ConsSE( QcSE("Y"),
                    ConsSE(ConsSE(ConsSE(
                                        QcSE("B2"),
                                        ConsSE(QcSE("S"),
                                               ConsSE(ConsSE(
                                                    ConsSE( QcSE("C1"),
                                                            QcSE("IF")),
                                                    ConsSE( QcSE("EQ"),
                                                            QIntSE(1))),
                                                    QIntSE(1)))),
                                      ConsSE( QcSE("S"),
                                              QcSE("MUL"))),
                                ConsSE(ConsSE( QcSE("C"),
                                               QcSE("B")),
                                    ConsSE(ConsSE( QcSE("C"),
                                                   QcSE("SUB")),
                                        QIntSE(1))))));

  Argument := Eval(FN);
  DispSE(Argument);
END fakt.
```

FN ist dabei eine globale Variable, die dem Laufzeitsystem angehört. Im Programm ist das der zu reduzierende kombinatorische Graph. Die Funktion **Eval** reduziert den kombinatorischen Graphen (FN).

Die nach jeder Reduktion durchgeführte Stackrekonstruktion ist ein großer Nachteil für eine effektive Arbeit.

Eine Verbesserung dieser Implementation zur Erzeugung prozeduralen Codes kann bei der Realisierung strikter Funktionen erreicht werden. Falls eine Funktion strikt ist, könnte sie in der prozeduralen Sprache (durch Compiler) realisiert und vom Graphreduktor aufgerufen werden. Damit wird die Graph-Reduktion der Graphkomponenten, die strikten Funktionen entsprechen, vermieden.

Eine solche Optimierung der auf kategorischen Kombinatoren basierenden Implementationstechnik wurde in [LL92] dargestellt. Mit dem in [LL92] angegebenen Algorithmus werden Konzepte wie z. B. Funktionalität aller Sprachkonstrukte (z. B. *if*), Funktionen höherer Ordnung, *curried functions* usw. jedoch nicht behandelt. Die Technik der SK-Graph-Reduktion unterscheidet sich sowohl von der Technik der auf kategorischen Kombinatoren basierenden abstrakten Maschine als auch von der Technik der abstrakten G-Maschine dadurch, daß bei der SK-Technik die Menge der Kombinatoren fest ist.

5.4 Eine für die Quelltexttransformation modifizierte SK-Graph-Reduktion

In diesem Kapitel wurde bisher beschrieben, wie man die auf SK-Graph-Reduktion basierende Implementation einer nicht-strikten funktionalen Sprache durch Übersetzung in eine prozedurale Sprache durchführen kann.

In diesem Abschnitt geben wir einen optimierenden Übersetzungsalgorithmus für die Erzeugung des Zielcodes in einer prozeduralen Sprache an. Die Optimierung basiert auf der Eigenschaft des Compilers, daß der Zielcode Code einer höheren Programmiersprache ist. Unsere Idee besteht darin, die „teure" Graph-Reduktion nur für die Anwendung der nicht-strikten Funktionen auszuführen. Alle strikten Funktionen werden direkt in die Zielsprache übersetzt, als Funktionen des Zielprogramms definiert und aus dem System für die Graph-Reduktion aufgerufen.

Sei die Quellsprache die in der Abbildung 3.2 auf Seite 50 im Kapitel 3 angegebene funktionale Sprache. Der Übersetzer soll die funktionalen Ausdrücke in zwei Mengen verteilen:

- Ausdrücke, die mit der im Kapitel 3 definierten Übersetzungsfunktion T übersetzt werden sollen (strikte Funktionen)

- Ausdrücke, die in kombinatorische Terme übersetzt werden sollen (sonstige Ausdrücke des funktionalen Quellprogramms)

Führen wir zunächst die für die Übersetzung geeigneten Funktionen ein.

5.4.1 Übersetzungsfunktionen

Folgende Funktionen werden eingeführt:

- $T_{qtt\_sk_{opt}}$ - die Hauptübersetzungsfunktion
- T^* - Funktion, die aus einem funktionalen Programm die Definitionen seiner strikten Funktionen ermittelt und alle Aufrufe der Funktionen in die entsprechende kombinatorische Form transformiert. Das Ergebnis der Funktion ist ein zweistelliges Tupel, dessen Komponenten aus den Definitionen der strikten Funktionen und dem Rest des funktionalen Programms (also das ursprüngliche funktionale Programm ohne strikte Funktionen) bestehen.
- *MkProgramm* - Funktion für die abschließende Erzeugung des prozeduralen Programms. Diese Funktion dient der Erzeugung des Programms einer konkreten prozeduralen Sprache.
- T_{qttSK} - die modifizierte im Kapitel 3 definierte Funktion für die Übersetzung einer funktionalen in eine prozedurale Sprache.
- C^* - modifizierte Funktion für die Übersetzung der funktionalen Quellsprache in kombinatorische Terme (siehe Abschnitt 2.4).

Die noch fehlenden genaueren Definitionen der o.g. Funktionen führen wir im folgenden Abschnitt ein.

5.4.2 Übersetzungsalgorithmus

Der Übersetzungsalgorithmus stützt sich auf den im Kapitel 3 beschriebenen Algorithmus für die Übersetzung einer funktionalen in eine prozedurale Sprache und auf einem Algorithmus für die Übersetzung einer funktionalen Sprache in kombinatorische Terme. Im Abschnitt 2.4 ist ein einfacher SK-Compilationsalgorithmus beschrieben. Die konkrete Implementation basiert auf einer etwas komplizierteren SK-Graph-Reduktion (größere Menge von Kombinatoren). An dieser Stelle werden wir die Quelltexttransformation mit der SK-Graph-Reduktion in Verbindung bringen. Die angewendeten SK-Compilationsalgorithmen (*bracket abstraction algorithms*) werden nicht näher analysiert.

Da die wesentlichen Algorithmen schon angegeben wurden (funktionale Sprache $\rightarrow$ prozedurale Sprache, funktionale Sprache $\rightarrow$ kombinatorischer Term), besteht unser Compilationsschema aus Aufrufen dieser Algorithmen bzw. einer Aufspaltung des funktionalen Quellprogramms in Teile, die in Kombinatoren bzw. Funktionen einer prozeduralen Sprache übersetzt werden sollen.

Definieren wir nun den Übersetzungsalgorithmus. Sei das zu übersetzende funktionale Programm mit *fp* bezeichnet.

$$T_{qtt\_sk_{opt}}[fp] = MkProgramm(\, T_{qttSK}[P_1],\; C^*[P_2])$$

wobei:

$$T^*[\, fp] = (P_1,\; P_2)$$

Definieren wir nun die o.g. Funktionen.

Funktion C^*

Mit C^* ist die Funktion für die Übersetzung eines funktionalen Ausdrucks in einen kombinatorischen Term bezeichnet. Die Definition der Funktion C^* unterscheidet sich von der im Abschnitt 2.4 definierten Funktion C nur bei der Übersetzung der Applikationen von strikten Funktionen:

$$C^*[e] = \begin{cases} \textbf{strictfun}\ Name\ C^*[(e_1 \ldots e_n)],\ falls\ e = (Name\ e_1 \ldots e_n)\ und \\ Name\ Identifikator\ einer\ strikten\ Funktion\ ist \\ \\ \\ C[e], \quad ansonsten \end{cases}$$

Mit $(e_1\ \ldots\ e_n)$ wird eine Liste mit den Elementen $e_1\ \ldots\ e_n$ bezeichnet. **strictfun** ist ein neuer zweistelliger Kombinator. **strictfun** *Name* $(e_1\ \ldots\ e_n)$ bezeichnet den Aufruf der Funktion *Name* mit den Argumenten $e_1\ \ldots\ e_n$. Definieren wir nun die Reduktionsregel für den Kombinator **strictfun**:

strictfun *Name* $(e_1\ \ldots\ e_n) \rightarrow$ **apply** *Name* $e_1\ \ldots\ e_n$

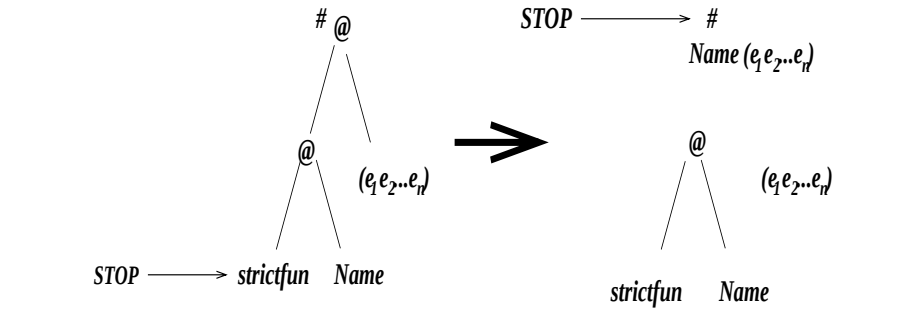
Dabei ist die Funktion *Name* eine direkt übersetzte Funktion der funktionalen Quellsprache in *Modula-2*. Mit **apply** *Name* $e_1\ \ldots\ e_n$ ist der Aufruf dieser Funktion mit den Argumenten $e_1\ \ldots\ e_n$ in der Zielsprache bezeichnet. In der Abbildung 5.2 ist die Reduktion des Kombinators **strictfun** dargestellt. Mit *STOP* wird der Anfang des *Spine*-Stacks bezeichnet.

Funktion T_{qttSK}

$$T_{qttSK}[e] \Leftrightarrow \mathrm{T\_}[prog,\ [],\ e,\ T_e,\ [],\ [],\ [],\ []]$$

Die Funktion T_ wird ähnlich der Funktion T aus dem Kapitel 3 definiert. Von der Funktion T unterscheidet sie sich nur bei der Erzeugung des Zielcodes:

Abbildung 5.2: Reduktion des Kombinators **strictfun**



$$\mathrm{T\_}[fun,\ FP,\ [],\ T_e,\ H_e\ ,\ H_v,\ H_p] \Leftrightarrow\!\!\rightarrow H_p$$

Die Funktion T_ erzeugt eine Funktion, im Gegensatz zur ursprünglichen Funktion T, die ein *QTTZ*-Programm liefert.

Funktion T^*

$$T^*\ (\begin{pmatrix} \textbf{letrec} & \\ & a_1 = e_1 \\ & a_2 = e_2 \\ \ldots & \\ & a_n = e_n \\ \textbf{in} & e \end{pmatrix}] \Leftrightarrow\!\!\rightarrow (\begin{pmatrix} \textbf{letrec} & \\ & a_{k_1} = e_{k_1} \\ & a_{k_2} = e_{k_2} \\ \ldots & \\ & a_{k_l} = e_{k_l} \\ \textbf{in} & () \end{pmatrix}, \begin{pmatrix} \textbf{letrec} & \\ & a_{k_j} = e_{k_j} \\ & a_{k_{j+1}} = e_{k_{j+1}} \\ \ldots & \\ & a_{k_{j+n-l-1}} = e_{k_{j+n-l-1}} \\ \textbf{in} & e \end{pmatrix})$$

Die Ausdrücke e_{k_1},e_{k_2}...e_{k_l} sind dabei die Lambda-Abstraktionen unter den Ausdrücken e_1, e_2 ... e_n, die strikte Funktionen sind, und e_{k_j}, $e_{k_{j+1}}$...$e_{k_{j+n-l-1}}$ sind die restlichen Ausdrücke.

Ansonsten gilt:

$$T^*[e] \Leftrightarrow\!\!\rightarrow ((), e)$$

Mit () bezeichnen wir den leeren funktionalen Ausdruck. Dieser Ausdruck tritt nur bei dem funktionalen Ausdruck, der direkt in die prozedurale Zielsprache zu übersetzen ist, als Ausdruck einer **letrec**-Konstruktion auf. In dem Falle werden nur

Funktionen in der Zielsprache erzeugt.

Funktion *MkProgramm*

Die Funktion *MkProgramm* ist die Funktion, die aus einem *QTTZ*-Programm und einem SK-kombinatorischen Term ein fertiges Programm in einer konkreten prozeduralen Sprache erzeugt. Mit der Funktion *MkProgramm* werden folgende Bestandteile ermittelt:

- Importliste
- Definition der globalen und lokalen Variablen
- Prozedurdefinitionen (aus dem von der Funktion T_{qttSK} gelieferten *QTTZ*-Programm, und - falls nötig - die Prozedur für die Aufrufe der strikten Funktionen aus dem SK-Reduktionssystem)
- Code für die Graph-Reduktion (siehe Abschnitt 5.3).

Eine formale Definition dieser Funktion geben wir nicht an. Die Funktion ist von der konkreten Zielsprache (bzw. vom Dialekt der Zielsprache) abhängig und nicht schwer zu definieren (die Transformation von *QTTZ* in *Modula-2* ist im Kapitel 3 definiert worden). Im folgenden Beispiel werden wir sehen, wie die Funktion *MkProgramm* in unserem System funktioniert.

Beispiel

Betrachten wir das folgende funktionale Programm:

```
e = letrec
        fakt  = lambda n. if (= n 1) then 1
                                     else (* fakt (- n 1))

        nth   = lambda n l. if (=  n  1) then (car l)
                                         else (nth (- n 1) (cdr l))

        prime = lambda n. (cons n (prime (+ n  1)))

    in (fakt (nth 10 (prime  5)))
```

Für die Funktionsapplikation haben wir wegen der einfacheren Schreibweise die Form (e_1 e_2 ... e_n) anstatt (**app** e_1 e_2 ... e_n) benutzt.

$$T_{qtt\_sk_{opt}}[\mathrm{e}] = MkProgramm(\ T_{qttSK}[P_1],\ C[P_2])$$

Die Funktion *fakt* ist eine strikte Funktion und wird direkt in die prozedurale Sprache übersetzt:

$$T^*[e] = \left(\begin{array}{ll} \textbf{letrec} & \\ & fakt\ =\ \textbf{lambda}\, n. \ldots \\ \textbf{in} & () \end{array} \right), \left[\begin{array}{ll} \textbf{letrec} & \\ & nth\ =\ \textbf{lambda}\, n\, l. \ldots \\ & prime\ =\ \textbf{lambda}\, n. \ldots \\ \textbf{in} & \textbf{strictfun}\ fakt\ (nth\ 10\ (prime\ 5)) \end{array} \right)$$

Daraus folgt:

$$P_1 = \left[\begin{array}{ll} \textbf{letrec} & \\ & fakt\ =\ \textbf{lambda}\, n. \ldots \\ \textbf{in} & () \end{array} \right] \text{ und}$$

$$P_2 = \left[\begin{array}{ll} \textbf{letrec} & \\ & nth\ =\ \textbf{lambda}\, n\, l. \ldots \\ & prime\ =\ \textbf{lambda}\, n. \ldots \\ \textbf{in} & \textbf{strictfun}\ fakt\ (nth\ 10\ (prime\ 5)) \end{array} \right]$$

P_1 wird mit der Funktion T_{qttSK} übersetzt:

$T_{qttSK}[\mathrm{P_1}] =$ (PROC:fakt:ARGS:n:VARS:BEGIN:IF:APP:=:ARG:n:ARG:1:ENDAPP:
THEN:1:ELSE:APP:*:ARG:n:APP:fakt:ARG:APP:-:ARG:n:ARG:1:
ENDAPP:ENDAPP:ENDAPP:ENDIF:ENDPROC)

P_2 wird in einen kombinatorischen Term übersetzt:

$C[P_2] =$

```
(("_C" U(("_C" B2."_C" U )("_C" B2 . "_C" K)"_C" STRICTFUN . FAKT)
(( "_C" C1 . "_C" C)(("_C" C1 "_C" B2 . "_C" CONS )("_C" C ."_C" I).5)
("_C" C."_C"I).5)) "_C" Y "_C" U ("_C" B ."_C" U)(("_C" C1 "_C" B2 . "_C" K)
(("_C" B2."_C" CONS) "_C" S("_C" C (("_C" B2 . "_C" S1). "_C" IF) "_C" EQ.1).
"_C" CAR) (("_C" C1 ."_C" C)("_C" C "_C" B2 . "_C" B) ("_C" C . "_C" SUB).1).
"_C" CDR) ("_C" C (("_C" B2 ."_C" CONS) "_C" S."_C" CONS)("_C" C . "_C" B)
"_C" ADD. 1))
```

Die beiden o.g. Programme sind nun mit der Funktion *MkProgramm* zusammenzusetzen. Diese Funktion stellt das fertige prozedurale Programm her. In der Abbildung 5.3 ist das erzeugte *Modula-2*-Zielprogramm dargestellt.

Implementation

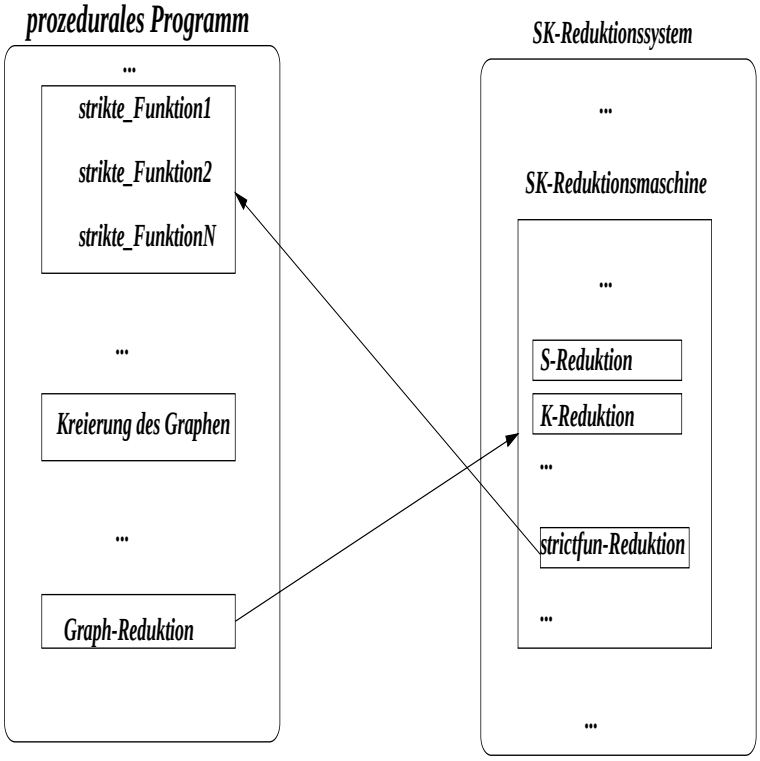
Die Implementation erinnert an die in [MB93] angegebene Implementation von *foreign functions* im SK-Graph-Reduktion-System. Die durch den Algorithmus aus Kap. 3 übersetzten Funktionen können für das SK-System als *foreign functions* be-

Abbildung 5.3: Erzeugtes Zielprogramm mit direkt übersetzten strikten Funktionen

```
MODULE beispiel;
FROM SExps IMPORT
  InitSE, HeadSE, TailSE, QIntSE,
  NullSE, DispSE, QsSE, ConsSE,
  ValsSE, QcSE, UpdateSE, ValIntSE;

FROM SExpsKr3 IMPORT
  MakeConstantSE,  SExp;

FROM SK_machine IMPORT
  Argument, MainProgram, ExecAdr,
  FN, ARG, FinalEval, Eval, MF;


FROM SYSTEM IMPORT TRANSFER, ADDRESS;

FROM InOut IMPORT
  WriteString, WriteLn, Write,
  WriteCard;

FROM SimpleIO IMPORT
  ReadString, ReadLn;

FROM SYSTEM IMPORT
  NEWPROCESS;

FROM Storage IMPORT
  ALLOCATE;


CONST
   WorkSpaceSize = 10000;

VAR
  WorkSpace: ADDRESS;

PROCEDURE FAKT (N:INTEGER):INTEGER;
BEGIN
  IF (N = 1) THEN
    RETURN 1
  ELSE
    RETURN (N * FAKT((N-1)))
  END;
END FAKT;


PROCEDURE EXSF;
VAR
  StrictFunIdent : INTEGER;
  StrictFunArgs  : SExp;
  Result         : SExp;
  Graph          : SExp;
  P, P1   : SExp;


BEGIN
  LOOP
    Graph:= Argument;(*!*)
    P1 := TailSE(HeadSE(Graph));(*!!*)
    P := FinalEval(Eval(P1)); (*!!!*)
    StrictFunIdent := ValIntSE(P); (*!!!!*)
    P1 := TailSE(Graph); (*!!!!!*)
    StrictFunArgs  := FinalEval(Eval(P1)); (*!!!!!!*)
    CASE StrictFunIdent OF
     1: Result:= QIntSE(FAKT(ValIntSE(
                              HeadSE(StrictFunArgs))));
        (* Aufruf *)

        UpdateSE(Graph, Result);TRANSFER(MF.Adres[1],
                                         ExecAdr) |
        (* Ersetzung*)

    END;
  END;
END EXSF;

BEGIN
  ALLOCATE(WorkSpace, WorkSpaceSize);
  NEWPROCESS(EXSF, WorkSpace,
             WorkSpaceSize, MF.Adres[1]);
  MF.No := 1;

  InitSE;
  MF.Names[1] := QsSE('EXSF');
  MakeConstantSE(MF.Names[1]);
  FN :=  ConsSE(ConsSE( QcSE("U"),ConsSE(ConsSE
  (ConsSE( QcSE("B2"), QcSE("U")),ConsSE(ConsSE
  ( QcSE("B2"), QcSE("K")),ConsSE(
  QcSE("STRICTFUN"), QIntSE(1)))),ConsSE(ConsSE
  (ConsSE( QcSE("C1"), QcSE("C")),ConsSE(ConsSE
  (ConsSE( QcSE("C1"),ConsSE( QcSE("B2"), QcSE
  ("CONS"))),ConsSE(ConsSE( QcSE("C"), QcSE("I")),
  QIntSE( 5))),ConsSE(ConsSE( QcSE("C"),
  QcSE("I")), QIntSE( 5)))),NullSE))),ConsSE
  (QcSE("Y"),ConsSE( QcSE("U"),ConsSE(ConsSE(
  QcSE("B"), QcSE("U")),ConsSE(ConsSE(ConsSE(
  QcSE("C1"),ConsSE( QcSE("B2"), QcSE("K"))),
  ConsSE(ConsSE(ConsSE( QcSE("B2"),
  QcSE("CONS")),ConsSE( QcSE("S"),ConsSE(
  ConsSE( QcSE("C"),ConsSE(ConsSE(ConsSE(
  QcSE("B2"), QcSE("S1")), QcSE("IF")),ConsSE
  ( QcSE("EQ"), QIntSE( 1)))), QcSE("CAR")))),
  ConsSE(ConsSE(ConsSE( QcSE("C1"), QcSE("C")),
  ConsSE(ConsSE( QcSE("C"),ConsSE( QcSE("B2"),
  QcSE("B"))),ConsSE(ConsSE( QcSE("C"),
  QcSE("SUB")), QIntSE( 1)))), QcSE("CDR")))),
  ConsSE(ConsSE( QcSE("C"),ConsSE(ConsSE(
  ConsSE(QcSE("B2"), QcSE("CONS")),ConsSE(
  QcSE("S"), QcSE("CONS"))),ConsSE(ConsSE(
  QcSE("C"), QcSE("B")),ConsSE( QcSE("ADD"),
  QIntSE( 1))))),NullSE))))));


  ARG:= NullSE;
  TRANSFER(MainProgram, ExecAdr);
  WriteString("Result:"); DispSE(Argument); WriteLn;
END beispiel.
```

trachtet werden. In der Abbildung 5.4 ist die Struktur des erweiterten SK-Systems dargestellt.

Abbildung 5.4: Struktur des erweiterten SK-Reduktionssystems



A $\Leftrightarrow\!\!\rightarrow$ **B** bedeutet „**A** *ruft Funktionen aus* **B** *auf*". So ruft beispielsweise die Graph-Reduktion des Zielprogramms die SK-Reduktionsmaschine und die strictfun-Reduktion der SK-Reduktionsmaschine die Funktionen *strikte_Funktion*K (K= 1 .. N) auf. Die Kommunikation zwischen dem SK-Graph-Reduktor und benutzer-definierten Funktionen erfolgt durch die globale Variable *Argument*. Die Programmkontrolle wird mit Hilfe des *Modula-2*-Befehls TRANSFER auf die Benutzerfunktionen im Zielprogramm umgeleitet.

Beschreiben wir nun die Einzelprozeduren für die Evaluation der strikten Funktionen im SK-Reduktionsmechnismus näher. In der Abbildung 5.5 ist die Funktion für die Reduktion des Kombinators *strictfun* angegeben.

Abbildung 5.5: Prozedur für die Reduktion des Kombinators *strictfun*

```
PROCEDURE RedStrictFun;
VAR p, p1, FunName: SExp;
    Ind: CARDINAL;
BEGIN
  FunName := QsSE("EXSF");
  IF STop-SBase < 3 THEN
     inWHNF := TRUE;
     RETURN
  END;
  Argument := Spine[STop-2].ex; (**)
  Ind := Ad(FunName); (***)
  TRANSFER(ExecAdr, MF.Adres[Ind]); (****)
  DEC(STop,2); (*****)
END RedStrictFun;
```

Die Funktion *EXSF* (*handler* für die strikten Funktionen) ist im Zielprogramm des Compilers erzeugt worden und ruft die strikten Funktionen auf, die als Funktionen in die prozedurale Zielsprache übersetzt worden sind. In der mit (**) bezeichneten Programmzeile wird der Variablen *Argument* die Liste der Argumente für die strikten Funktionen zugeordnet. In der Zeile (***) wird die Adresse der Funktion, in der die strikten Funktionen aufgerufen werden, ermittelt. In (*****) wird diese Funktion ,,aufgerufen". Die Funktion EXSF nimmt den ganzen Graphen (siehe Abbildung 5.3, Programmzeile (*!*)), evaluiert den Identifikator der strikten Funktion ((*!!!*)und (*!!!!*)) und ihre Argumente ((*!!!!!*) und (*!!!!!!*)). Die strikte Funktion wird danach aufgerufen (einer der CASE-Fälle). In unserem Beispiel gibt es nur eine strikte Funktion, so daß CASE nur einen Fall enthält (*Aufruf*). Daran anschließend wird die Wurzel des Graphen durch das Ergebnis der Funktionsanwendung ersetzt ((*Ersetzung*)).

Laufzeitvergleich

Die Tabelle in der Abbildung 5.6 stellt den Laufzeitvergleich der Programme der optimierten ($SK_{qtt_{opt}}$) mit der nicht-optimierten(SK_{qtt}) SK-Maschine bei der Erzeugung von *Modula-2*-Code dar.

Die erreichten Effektivitätsverbesserungen sind erheblich. Besonders groß ist der Gewinn in den Fällen, wo es sich um rein-numerische Programme handelt (z. B. **fib, tak**), weil das funktionale Programm transparent in ein prozedurales Programm ohne zusätzliche Datenstrukturen und Manipulationen einer abstrakten Maschine (alle Funktionen sind strikt!) übersetzt wird.

Wir haben ein Verfahren der Implementation einer nicht-strikten funktionalen Sprache durch SK-Graph-Reduktion mit einer Erzeugung von Zielcode in einer pro-

Abbildung 5.6: Laufzeitvergleich

	SK_{qtt} $[\mu s]$	$SK_{qtt_{opt}}$ $[\mu s]$	Vergleich
(diff (plus (mal x x) 2))	7.404,00	2.137,80	SK_{qtt} -71,13%
(fib 20)	9.296.882,10	16.909,60	SK_{qtt} -99,82%
(fakt 20)	7.966,80	1.403,30	SK_{qtt} -82,39 %
(queen 8)	5.745.098,90	238.268,20	SK_{qtt} -95,85%
(tak 30 25 19)	18.264.202,90	2.786.969,60	SK_{qtt} - 84,74 %
union	130.140,10	22.646,80	SK_{qtt} -82,60%

zeduralen Sprache beschrieben. Die in den SK-Kombinatoren modellierte Graph-Reduktion ist ungeeignet für die Erzeugung von prozeduralem Zielcode. Durch die von uns angegebene Erweiterung des SK-Systems wird diese Technik verbessert, falls im Quellprogramm strikte Funktionen erscheinen.

Das Verfahren kann mit der auf der abstrakten G-Maschine basierenden Technik verglichen werden. Bei dieser Technik wird das funktionale Quellprogramm durch λ-Lifting in Funktionen ohne innere λ-Abstraktionen übersetzt. Die Funktionen werden weiter compiliert und ,,warten" auf ihre Anwendungen.

In unserem Modell werden nicht-strikte Funktionen in Funktionen der prozeduralen Zielsprache direkt übersetzt. Im Evaluationsprozeß werden sie aus dem SK-Simulator durch den *handler* für strikte Funktionen (Funktion *EXSF* in der Abbildung 5.3) aufgerufen. Dabei basiert der SK-Simulator auf einer festen Kombinatorenmenge.

Kapitel 6

Zusammenfassung und Ausblick

In der vorliegenden Dissertation wurde die Quelltexttransformation als spezielle Implementationsstrategie für funktionale Sprachen untersucht. Der Zielcode eines auf der Quelltexttransformation basierenden Compilers ist ein Programm einer prozeduralen Sprache. Zwei grundlegende Methoden der Übersetzung wurden bearbeitet:

1. Direkte Übersetzung funktionaler in prozedurale Sprachen und
2. Modifikation bzw. Erweiterung existierender Implementationstechniken im Hinblick auf die Erzeugung des Zielprogramms in einer prozeduralen Sprache.

Folgende *grundlegende Ergebnisse* der Dissertation sollen hervorgehoben werden:

a) Existierende Ansätze zur Quelltexttransformation funktionaler Sprachen [oCS93, Bar93, Tam93, YU93, Aug84, Aug93, TAL90, DFG+94, ACD+97, CSS95, Jä96] wurden aufbereitet, zusammengefaßt, systematisiert und verglichen.

b) Die Technik der Quelltexttransformation wurde mit herkömmlichen Implementationstechniken [Lan64, Wad71, Dil88, Aug84, Jon87, San95] funktionaler Sprachen kombiniert.

c) Zur Implementation funktionaler Sprachen über eine **direkte** Transformation in eine prozedurale Sprache wurde ein formales System von Transformationsregeln aufgestellt. Hierfür wurden bekannte Techniken wie das Lambda-Lifting [Aug84, Jon87] und das Transformationssystem von Santos [San95] ebenso einbezogen wie darüber hinausgehende selbst entwickelte spezielle Regeln.

d) Auf dieser theoretischen Grundlage haben wir eine Implementation vorgenommen, bei der beispielhaft eine Teilsprache von *Scheme* [RE91] (erweiterter Lambda-Kalkül [Bar84]) nach Modula-2 [Wir85] übersetzt wird.

e) Der von uns implementierte Compiler besitzt im Vergleich zu einer im Ansatz ähnlichen Implementation von Bartlett [Bar93] eine Reihe von Vorteilen:

- Zusammenhänge zwischen Quell- und Zielsprache sind transparenter gestaltet.
- Erzeugte Typen werden so speziell wie möglich übersetzt, in dem die Typenvielfalt prozeduraler Sprachen ausgenutzt wird.
- Unser Compiler arbeitet (aufgrund o. g. Eigenschaften) z. T. erheblich effektiver als vergleichbare Implementationen.

f) Im Vergleich mit Compilern, die auf dem *Continuation-Passing-Style* basieren [App92, KKR+86, TAL90], ist folgender Unterschied zu nennen:
Transparente Zusammenhänge zwischen Quell- und Zielsprache werden mit unserem Compiler hergestellt. Dabei werden insbesondere prozedurale Programme erzeugt, die mehr dem prozeduralen Programmierparadigma entsprechen, während der *Continuation-Passing-Style* Programme produziert, deren Hintergrund noch das funktionale Paradigma darstellt.

g) Für die herkömmliche Technik der Stack-basierten Implementation (SECD-Maschine) [Lan64] wurden zwei Möglichkeiten der Kombination mit den Ansätzen der Quelltexttransformation aufgezeigt. Dabei besitzt die Implementation ohne ein C-Register Effektivitätsvorteile.

h) Für die Graph-Reduktion-basierte Implementationstechnik wurde eine Kombinationsmöglichkeit mit der Quelltexttransformation untersucht, wobei sich ebenfalls eine erhöhte Effektivität, Transparenz und Lesbarkeit der erzeugten Programme ergab. Eine Implementation der vorgeschlagenen Strategie wurde von uns realisiert.

Als *offene bzw. weiterführende Probleme* unserer Dissertation sind folgende Aspekte zu nennen:

- Verkettung von Quelltextcompilern: Auf der Grundlage einer Implementation der Quelltexttransformation einer funktionalen Sprache (z. B. *Scheme*) in eine prozedurale Sprache (z. B. *Modula-2*) kann die Kombination mit weiteren Quelltextcompilern sowohl als Front-End als auch als Back-End vorgenommen werden.

  Beispielsweise könnte der Miranda^{TM}-Scheme-Compiler [Jør92] mit dem Scheme-Modula-2-Compiler zu einem Miranda^{TM}-Modula-2-Compiler kombiniert werden. Auf der Seite des Back-Ends wiederum könnte mit Hilfe eines Modula-2-C-Compilers [Pri92] ein Scheme-C-Compiler entstehen. Und durch Kombination der drei genannten Compiler wäre ein Miranda^{TM}-C-Compiler denkbar. Hier müssen allerdings erst Experimente Aufschluß geben, inwiefern Probleme der jeweiligen Laufzeitsysteme aufwendig zu lösen sind.

- Einbeziehung eines Debugging auf der Ebene der Quellsprache: Nutzt man einen Quelltextcompiler als Produktionscompiler, so muß zur Fehlersuche das Laufzeitverhalten (z. B. Ausnahmesituation) mit den Sprachelementen des Quellprogramms verbunden werden (z. B. Hinweis auf Zeilennummern des Quellprogramms). Hier sollten Experimente über geeignete Maßnahmen zu Erfahrungen führen.

- Unterstützung der einfachen Typen der Zielsprache bei der auf der abstrakten SECD-Maschine basierenden Implementationstechnik: In der vorliegenden Dissertation wurde ein Algorithums für die Übersetzung des SECD-Codes in eine prozedurale Sprache angegeben. Die Erzeugung von Programmen mit einfachen Typen der Zielsprache ist dabei nicht analysiert worden.

- Realisierung einer Kombination der im Kapitel 3 betrachteten direkten Übersetzung einer funktionalen Sprache in eine prozedurale Sprache mit der abstrakten SECD-Maschine: Direkte Übersetzung der Konstrukte der funktionalen Quellsprache, die in eine prozedurale Sprache direkt übersetzt werden können und die Realisierung der nicht direkt übersetzbaren Elemente der funktionalen Sprache durch die abstrakte SECD-Maschine. Hierzu müssen (wie im Kapitel 5) zusätzliche Übersetzungsfunktionen und Verfahren für die Kommunikation zwischen den in der prozeduralen Sprache direkt erzeugten Funktionen und der SECD-Maschine definiert werden. Eine derartige Implementation kann zu einem Gewinn an Lesbarkeit und Debuggingmöglichkeit der erzeugten Programme führen.

- Eine Möglichkeit der Optimierung des auf der SK-Graph-Reduktion basierenden Quelltextcompilers wäre eine Vereinfachung des Prozesses der Suche des zunächst zu reduzierenden Knotens im kombinatorischen Term. Die nur für nicht-strikte Funktionen des funktionalen Quellprogramms durchzuführende Graph-Reduktion wäre damit beschleunigt. Mit diesem Thema beschäftigt sich z. B. die Arbeit von Fradet und Metayer [FM91].

- Ein Vergleich der in der Arbeit definierten modifizierten SK-Maschine mit der abstrakten G- bzw. STG-Maschine wäre interessant. Um den Einfluß unterschiedlicher Compiler (für *Modula-2* bzw. *C*) auf die Vergleichswerte auszuschließen, ist eine Reimplementierung unseres Compilers zur Erzeugung von *C*-Zielcode erforderlich. Die einzigen uns bekannten auf diesen abstrakten Maschinen basierenden Quelltextcompiler haben als Zielsprache die Sprache *C*.

- Das definierte Verfahren für die Durchführung der Stack-Manipulation im prozeduralen Programmierparadigma ist auf die Stack-Manipulation der abstrakten G-Maschine übertragbar: Die G-Maschinenbefehle für die Stack-Manipulation (mit dem *handler* für rekursive Funktionen, mit *direkter* Übersetzung

der Maschinenbefehle, die ihr Pendant in der prozeduralen Zielsprache haben usw.), sind im prozeduralen Modell realisierbar.

Das definierte Verfahren für die Implementation der SK-Graph-Reduktion im prozeduralen Modell kann auch bei der Graph-Reduktion der abstrakten G-Maschine angewendet werden: Die Graph-Reduktion der G-Maschine kann auf die gleiche Art in einem prozeduralen Programmodell (mit dem *handler* für die Aufrufe der direkt übersetzten Funktionen aus dem G-Evaluator usw.) realisiert werden.

Die Implementation einer auf o. g. Aspekte basierenden abstrakten G-Maschine mit Unterstützung der *unboxed* Datenobjekte und deren Vergleich mit schon realiserten Ansätze [PJ92, Aug84] kann zu weiterführenden Ergebnissen führen.

Literaturverzeichnis

[ACD+97] P. K. T. Au, M. M. T. Chakravarty, J. Darlington, Y. Guo, S. Jähnichen, G. Keller, M. Köhler, M. Simons, and W. Pfannenstiel. Enlarging the scope of vector-based computations: Extending Fortran 90 with nested data parallelism. In W. Giloi, editor, *International Conference on Advances in Parallel and Distributed Computing*. IEEE Computer Society, 1997.

[ACPR94] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Didier Remy. Dynamic typing in polymorphic languages. Technical Report DEC-SRC-120, Digital Equipment Corporation, Systems Research Centre, Jan 94.

[AF71] Martin William A. and Richard J. Fateman. The MACSYMA system. In *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, ACM SIGSAM*, pages 59–75, 1971.

[AJ89] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the $<\nu$,g$>$-machine. In *Functional Programming & Computer Architecture*, pages 202–214. ACM, 89.

[AM91] A. Aiken and B. Murphy. Static type inference in a dynamically typed language. In *Principles of Programming Languages*, pages 279–290, January 1991.

[App92] A. W Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[Aug84] L. Augustsson. A compiler for lazy ML. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin*, pages 218–27, August 1984.

[Aug93] Lennart Augustsson. Haskell B. User's manual version 0.999.4, 1993.

[AWW90] Alexander S. Aiken, John H. Williams, and Edward L. Wimmers. Program transformation in the presence of errors. In *Principles of Programming Languages*, pages 210–217, January 1990.

[AWWar] A. Aiken, J. H. Williams, and E. Wimmers. Safe—a semantic technique for transforming programs in the presence of errors. *ACM Transactions on Programming Languages and Systems*, To Appear.

[Bar84] H. P. Barendregt. *The lambda Calculus, its Syntax and Semantics.* North Holland, Amsterdam, 1984.

[Bar93] Joel Bartlett. Scheme (Scheme to C compiler) ver. 15mar1993. Technical report, Digital Western Research Laboratory, 1993.

[Bar94] David T. Barnard. A simple functional language compiler. Technical Report 94-371, Queen's University, Department of Computing and Information Science, October 1994.

[BBM+85] F.L. Bauer, M. Broy, B. Möller, P. Pepper, M. Wirsing, et al. *The Munich Project CIP. Vol. I: The Wide Spectrum Language CIP-L.* Number 183 in Lecture Notes on Computer Science. Springer Verlag, Berlin, Heidelberg, New York, Berlin, 1985.

[BEH+87] F.L. Bauer, H. Ehler, R. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP. Vol. II: The Transformation System CIP-S, LNCS 292*, volume II. Springer Verlag, Berlin, Heidelberg, New York, Berlin, 1987.

[BF82] Klaus J. Berkling and Elfriede Fehr. A consistent extension of the lambda-calculus as a base for functional programming languages. *Information and Control*, 55(1–3):89–101, October/November/December 1982.

[BHHW89] K. Bothe, B. Hohberg, Ch. Horn, and O. Wikarski. A Portable High-Speed Pascal to C Compiler. In *SIGPLAN Notices*, September 1989.

[Bla93] R. Blach. Objektorientierung und Übersetzerbau. In *Tag der Informatik an der Humboldt-Universität zu Berlin*, Dez. 1993. Preprint Nr. 27.

[Blo89] A. G. Bloss. *Path Analysis and the Optimization of Non-strict Functional languages.* PhD thesis, Research Report YALEU/DCS/RR-704, Yale University, 1989.

[BMI93] Z. Budimac, D. Macos, and M. Ivanovic. Jedan algoritam za prevodjenje lambda-izraza u term kombinatorske logike. In *VIII konferencija za primenjenu matematiku*, Tivat, 1993.

[BMI94] Z. Budimac, D. Macos, and M. Ivanovic. Another bracket abstraction algorithm. In *In Proc. of 8. Conference on Applied Mathematics*, Tivat, Yugoslavia, 1994.

[BMS80] R. M. Burstall, D.B. MacQueen, and D. T. Sannella. HOPE:An experimental applicative language. In *The 1980 LISP Conference*, pages 136–143, The USP Co., 1980. Stanford University, Santa Clara University.

[BN89] Mark R. Brown and Greg Nelson. IO streams: Abstract types, real programs. Technical Report 53, Digital Equipment Corporation, Systems Research Centre, nov 1989.

[Boq95] Urban Boquist. Interprocedural register allocation for lazy functional languages. In *Functional Programming & Computer Architecture*, La Jolla, California, June 1995.

[Bot90] K. Bothe. A prolog space benchmark suite: a new tool to compare prolog implementations. In *SIGPLAN Notices*, 12. 1990.

[Bot92a] K. Bothe. Meta-interpretation and partial evaluation in logic programming: An assessment of the power in real applications. In *Workshop 'Sprachen für KI-Anwendungen, Konzepte - Methoden - Implementierungen*, Bad Honnef, Mai 1992.

[Bot92b] K. Bothe. Prolog space benchmarking. In *Newsletter of the ALP.*, January 1992.

[Bot93] K. Bothe. Labor für Quelltexttransformation: Ziele und Ansätze. - Antrittsvorlesung. In *Tag der Informatik an der Humboldt-Universität zu Berlin*. Humboldt-Universität zu Berlin, Dez. 1993. Preprint Nr. 27.

[Bot95] K. Bothe. Compilation von Pascal nach C: Transformationsregeln. Technical report, Humboldt-Universität zu Berlin, 1995. Preprint.

[BW92] Richard Bird and Philip Wadler. *Einführung in die funktionale Programmierung*. Carl Hanser Verlag and Prentice-Hall International, 1992.

[Car93] Bryan Carpenter. Some lattice-based scientific problems, expressed in Haskell. Technical Report CSTR 93-06, Department of Electronicsand Computer Science, University of Southampton, March 93.

[Cas95] G. Castagna. Integration of parametric and "ad hoc" second order polymorphism in a calculus with subtyping. *Formal Aspects of Computing: International Journal of Formal Methods*, 1995.

[CF58] H.B. Curry and R Feys. *Combinatory logic, Vol.1,*. North-Holland, 1958.

[CGar] D.B. Carpenter and H. Glaser. Some lattice-based scientific problems expressed in Haskell. *Journal of Functional Programming*, to appear.

[Chu33] A. Church. A set of postulates for the foundation of logic. *Ann. Math 2, 33-34, 346-366, 839-864*, 1932-1933.

[Chu41] A. Church. The calculi of lambda conversion, 1941.

[CR36] A. Church and J.B Rosser. Some properties of conversion. *Trans. Am. Math. Soc. 39*, pages 472–482, 1936.

[Cro88] D. Crookes. Translation as a language implementation technique for supercomputers. *Software Engineering Journal*, 3(2), 1988.

[CSS95] Manuel M.T. Chakravarty, Friedrich Wilhelm Schröer, and Martin Simons. V–nested parallelism in c. In *Proceedings of the Working Conference on Massively Parallel Programming Models, IEEE Computer Society*, 1995.

[Cun94] H. Conrad Cunningham. Formal methods in functional programming. In *Teaching Formal Methods Curriculum Development Workshop*, August 94.

[DB72] N.G. De Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae. Vol.34*, pages 381–92, 1972.

[DB94] Dominic Duggan and Frederick Bent. Explaining type inference. Technical Report CS-94-14, University of Waterloo, Department of Computer Science, 1994.

[DC95] Roberto Di Cosmo. *Isomorphisms of types: from lambda-calculus to information retrieval and language design*. Progress in theoretical computer science. Birkhauser, 1995.

[DF90] Ernst-Erich Doberkat and Dietmar Fox. *Praktischer Übersetzerbau*. B.G. Teubner Stuttgart, 1990.

[DFG+94] Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In Jürg Gutknecht, editor, *Programming Languages and System Architectures, International Conference, Zurich, Switzerland, March 1994*, LNCS 782, pages 228–244. Springer, 1994.

[Dil88] A. Diller. *Compiling Functional Languages.* John Wiley and sons, Chichester, 1988.

[DJHT88] Y. Siret Davenport J. H. and E. Tournier. *Computer Algebra : Systems and Algorithms for Algebraic Computation.* Academic Press, 1988.

[DRW95] Catherine Dubois, Francois Rouaix, and Pierre Weis. Extensional polymorphism. In *Principles of Programming Languages*, 1995.

[Dze94] Hsianlin Dzeng. Type reconstruction for variable-arity procedures. In *Conference on Lisp and Functional programming*, pages 239–249, 1994.

[Efr94] Sofoklis Efremidis. On program transformations. Technical Report TR94-1434, Cornell University, Computer Science Department, June 1994.

[EG93] Sofoklis Efremidis and David Gries. An algorithm for processing program transformations. Technical Report TR93-1389, Cornell University, Computer Science Department, October 1993.

[eul94] Abschlußbericht des Verbundvorhabens APPLY, 1994.

[EW88] Margaret H. Eich and David L. Wells. Database concurrency control using data flow graphs. *j-TODS*, 13(2):197–227, jun 1988.

[Feh84] Elfriede Fehr. Expressive power of typed and type-free programming language. *Theoretical Computer Science*, 33(2+3):195–238, 1984.

[Feh89] Elfriede Fehr. *Semantik von Programmiersprachen.* Springer, Berlin, 1989.

[FG93] Alice E. Fischer and Frances S. Grodzinsky. *The Anatomy of Programming Languages.* Prentice Hall, 1993.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[FHB94] Stephen Fitzpatrick, T.J. Harmer, and J.M. Boyle. Deriving efficient parallel implementations of algorithms operating on general sparse matrices using automatic program transformation. In Bruno Buchberger and Jens Volkert, editors, *Parallel Processing: CONPAR 94-VAPP VI*, pages 148–159, Belfast BT7 1NN, Northern Ireland, sep 1994. Department of Computer Science, Queen's University of Belfast, Springer-Verlag.

[FM91] Pascal Fradet and Daniel Le Metayer. Compilation of functional languages by program transformation. In *ACM Transactions on Programming Languages and Systems. Vol13, No. 1*, pages 21–51, January 1991.

[FO95] Steve Fitzgerald and Rodney R. Oldehoeft. Update-in-place analysis for true multidimensional arrays. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 105–118, April 1995.

[FPL89] *ACM Conference on Functional Programming Languages and Computer Architecture*, London, September 1989.

[Fra88] Pascal Fradet. *Compilation des langages fonctionnels par transformation de programmes.* PhD thesis, l'Université de Rennes I, 1988.

[FW86] J. Fairbairn and S. Wray. Code generation techniques for functional languages. In *Proc. of the ACM Conference on Lisp and Functional Programming, Boston*, pages 94–104, 1986.

[FW87] J. Fairbairn and S. Wray. TIM: A simple lazy abstract machine to execute superkombinators. In *Proc. of Conference on Functional Programming and Computer Architecture*, 1987.

[GBH+96] Wolfgang Goerik, Harold Borey, Ulrich Hoffmann, Markus Perling, and Michael Sintek. Komplettkompilation von lisp: eine studie zur übersetzung von lisp-software für c-umgebungen. In *Künstliche Intelligenz, 20. Deutsche Jahrestagung für Künstliche Intelligenz*, pages 31–33, 1996.

[GED93] VW GEDAS. Babylon-system, 1993.

[GGR94] Lal George, Florent Guillame, and John H. Reppy. A Portable and Optimizing Back End for the SML/NJ Compiler. In *International Conference on Compiler Construction*, pages 83–97, April 1994.

[GH86] H. Glaser and S. Hayes. Another Implementation Technique for Applicative Languages. In *Proceedings of ESOP'86 - European Symposium on Programming, SV*, LNCS, pages 70–81, March 1986.

[GMM+78] M. J. C. Gordon, A. J. R. G. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth. A metalanguage for interactive proof in LCF. In *Fifth ACM Symp. on Principles of Programming Languages.* ACM Press, New York, 1978.

[GMW79] M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF.* Springer-Verlag LNCS 78, Berlin, 1979.

[Gol94] Benjamin Goldberg. Functional Programming Languages (tutorial). In *PLDI '94*, pages 1–64. ACM Sigplan, 1994.

[Goo90] F. Goodman. Fortran to C. *Journal of C Language Translation*, 2(2), September 1990.

[GP84] Allen Goldberg and Robert Paige. Stream processing. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 53–62. ACM, ACM, August 1984.

[GS84] Buchanan Bruce G. and Edward Hance Shortliffe. *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, 1984.

[Hal94] Cordelia V. Hall. Using Hindley Milner type inference to optimise list representation. In *Conference on Lisp and Functional programming*, 1994.

[Har94] P. H. Hartel. Benchmarking implementations of lazy functional languages ii – two years later. Technical Report Cs-94-21, Dept. of Comp. Sys, Univ. of Amsterdam, December 1994.

[Hen80] Peter Henderson. *Functional Programming Application and Implementation*. Prentice Hall, 1980.

[Hen93] M. C. Henson. Transformations as proofs. Technical Report CSM-195, University of Essex, Department of Computer Science, November 1993.

[HF92] Paul Hudak and Joseph H. Fasel. A Gentle Introduction to Haskell. In *ACM SIGPLAN Notices, Volume 27, No.5*, May 1992.

[Hin92] Ralf Hinze. *Einführung in die funktionale Programmierung mit Miranda*. B.G. Teubner Stuttgart, 1992.

[HJ94] Fritz Henglein and Jesper Jorgensen. Formally optimal boxing. In *Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[HK93] Wolfgang Goerik Heinz Knutzen, Ulrich Hoffman. Common Lisp, CLiCC (Common Lisp to C compiler) ver. 0.6.1, 1993.

[HL94] Ronert Harper and Mark Lillibridge. Polymorphic type assignment and cps conversion. *Lisp and Symbolic Computation*, 94.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators. Submitted for publication, January 1996.

[HM94] Robert Harper and Greg Morrisett. Compiling with non-parametric polymorphism (preliminary report). Technical Report CMU-CS-FOX-94-03, Carnegie Mellon University, Department of Computer Science, 94.

[HMB96] Therese Hardin, Luc Maranget, and Pagano Bruno. Functional backends whithin the lambda-sigma-calculus. In ACM press, editor, *Iternational Conference on Functional Programming*, May 1996.

[How92] Denis Howe. Miranda to Haskell Compiler. Technical report, Department of Computing, Imperial College, London, UK, 92.

[HPJWe92] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler (editors). Report on the Programming Language Haskell, a Non-strict Purely Functional Language (version 1.2). *SIGPLAN Notices*, Mar, 1992.

[HR95] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA), La Jolla, California.* ACM, ACM Press, June 1995.

[Hud89] Paul Hudak. *Conception, Evolution, and Application of Functional Programming Languages.* ACM Computing Surveys, 1989.

[Hug84] R.J.M. Hughes. *The design and implementation of programming languages.* PhD thesis, Programming Research Group, Oxford, September 1984.

[HWB94] Kevin Hammond, Philip Wadler, and Donald Brady. Imperate: Be imperative. Technical report, Department of Computing Science, University of Glasgow, 94.

[ICSIiB92] CA International Computer Science Institute in Berkley. Sather programming language and environment, ver. 0.2i. Technical report, International Computer Science Institute in Berkley, 1992.

[Ive62] K. Iverson. *A Programming Language.* Wiley, New York, 1962.

[Jä96] Stefan Jähnichen. Arbeiten zur Parallelverarbeitung. Humboldt-Universität zu Berlin, Institut für Informatik, Forschungskolloquium, Vortrag, Januar 1996.

[JeS94] Simon Peyton Jones and André Santos. Compilation by Transformation in the Glasgow Haskell Compiler, 1994.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. *Proceedings of the ACM Conference on Compiler Construction, Montreal*, pages 58–69, June 1984.

[Jon87] Simon L. Peyton Jones. *The implementation of functional languages.* Prentice Hall, 1987.

[Jon92] Mark P. Jones. A theory of qualified types. In *European symposium on programming, ESOP '92*, Rennes, France, February 1992. Springer Verlag LNCS 582.

[Jon94] Mark P. Jones. ML typing, explicit polymorphism and qualified types. In *In Proceedings of TACS '94*, Sendai, Japan, 94. Springer Verlag, LNCS 789.

[Jør92] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, New Mexico*, pages 258–268, January 1992.

[Joy] Mike Joy. The translation of high-level functional languages to flic. Technical Report RR142, University of Warwick, Computer Science.

[Jr.78] Guy L. Steele Jr. Rabbit: A C compiler for Scheme. Technical report, Massachusetts Institute of Technology, May 1978. Technical Report AI Memo No. 452.

[JS89] Peyton S. L. Jones and J. Salkid. The spineless tagless G-machine. In FPLCA [FPL89].

[KF92] Andreas Kind and Horst Friedrich. A practical approach to type inference for eulisp. *Lisp and Symbolic Computation*, 6(1/2):159–176, 92.

[KKR+86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. Orbit: An optimizing compiler for scheme. In *SIGPLAN Symposium on Compiler Construction*, pages 219–233, July 1986.

[Kop88] Herbert Kopp. *Compilerbau.* Carl Hanser Verlag, 1988.

[KW94] A. J. Kfoury and J. B. Wells. Adding polymorphic abstraction to ml (detailed abstract). Technical Report 94-006, Boston University, May 94.

[Lan64] P.J. Landin. The mechanical evaluation of expression. In *C Journal. Vol. 6*, pages 308–320, 1964.

[Lau93] Peter Lauer. Functional programming, concurency, simulation and automated reasoning, 1993.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Principles of Programming Languages*, pages 177–188, 1992.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Principles of Programming Languages*, pages 109–122, 1994.

[LJ94] John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *Sigplan '94, Conference on Programming Language Design and Implementation*, Orlando, Florida, June 10-24 1994.

[LL96] Mark Leone and Peter Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, February 1996.

[LL92] Rafael D Lins and Bruno O Lira. GCMC: A Novel Way of Compiling Functional Languages. Technical Report 19-92, Computing Laboratory, University of Kent, Canterbury, UK, July 92.

[LPJ94] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press.

[LT95] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, February 1995.

[LTPJ92] Raphael Lins, Simon Thompson, and Simon Peyton Jones. On the equivalence between cmc and tim. Technical Report 9-92, Computing Laboratory, University of Kent, Canterbury, UK, Nov 92.

[Mac94] Dragan Macos. Implementation of a Scheme to Modula-2 Translator. Technical report, Humboldt-University, Berlin, Dept. of Computer Science, Lindenstraße 54a 10099, Berlin, Germany, 1994.

[Mac95] Dragan Macos. Eine abstrakte Datenstruktur für die Implementation der nicht-strikten Auswertung in einem prozeduralen Programmierparadigma. Humboldt Universität, Berlin. Arbeitspapier, 1995.

[Mac96a] Dragan Macos. Implementation funktionaler Programmiersprachen durch eine ,,source to source" Transformation. Humboldt Universität, Berlin. Arbeitspapier, 1996.

[Mac96b] Dragan Macos. Typing a Functional Language in Procedural Paradigm. In *ACM State of the Art Summer School, Functional and Object Oriented Programming*, Sobotka, Poland, September 1996. Vortrag.

[Mac97] D. Macos. Implementation funktionaler programmiersprachen durch quelltexttransformation (Vortrag). In *Tag der Informatik an der Humboldt-Universität zu Berlin*. Humboldt-Universität zu Berlin, 1997.

[Mak91] Ronald Mak. *Writing Compilers and Interpreters:An Applied Approach*. John Wiley and Sons, Inc, 1991.

[Mar94] Luc Maranget. Two techniques for compiling lazy pattern matching. Research Report 2385, INRIA Rocquencourt, October 1994.

[Mas94] Dave Mason. The challenges of compiling sml systems code, 1994.

[Mau91] M. Mauny. Functional programming using caml. Technical Report RT-0129, National Institute for Research in Computer and Control Sciences (INRIA), 91.

[MB92a] D. Macos and Z. Budimac. Sk machine simulator, ver. nuam/sk. In Print, 1992.

[MB92b] D. Macos and Z. Budimac. Some experiences with sk-reduction machine - user's perspective. In *Abstracts of the VI Conference on Logic and Computer Science LIRA '92*, page 141, Novi Sad, October 1992.

[MB93] D. Macos and Z. Budimac. An implementation of foreign function in a lazy functional programming language. In *XXXVII conference of Etan*, Ulcinj, 1993.

[MB97] D. Macos and Z. Budimac. Implementing lazy foreign functions in a procedural programming language. In *VIII International Conference on Logic and Computer Science*, Novi Sad, September 1997.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM3, 4*, pages 184–195, 1960.

[McC63] J. McCarthy. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, pages 33–70, 1963.

[McC78] J. McCarthy. History of Lisp. *Preprints of Proceedings of ACM SIGPLAN History of Programming Languages Conference*, pages 217–223, 1978.

[MGRA93] D.B. MacQueen, Lal George, AJ. H. Reppy, and A. W. Appel. SML/NJ (Standard ML of New Jersey), ver. 0.93. New Jersey, 1993.

[Mic89] Greg Michaelson. *An Introduction to functional Programming through Lambda Calculus.* Addison Wesley, 1989.

[Mil78] Robin Milner. A theory of type polymorphism in programming. In *Journal of computer and system sciences 17*, pages 348–375, 1978.

[Mil84] Robin Milner. A Proposal for Standard ML. In *Symposium on LISPand Functional Programming*, pages 184–97. ACM Press, New York, 1984.

[ML92] Martin A Musicante and Rafael D Lins. GMC A Graph Categorical Multi-Combinator Machine. Technical Report 27-92, Computing Laboratory, University of Kent, Canterbury, UK, Nov 92.

[Mos75] Joel Moses. A macsyma primer. Technical report, Mathlab Memo no. 2, 1975.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambrige, MA, 1989.

[oCS91] School of Computer Science. Standard ml compiler to c. Technical report, Carnegie Mellon University, 1991.

[oCS93] Department of Computing Science. The Glasgow Haskell Compiler – version 0.19. Technical report, Glasgow University, Glasgow G12 8QQ, Scotland, 1993.

[Owe87] G. S. Owen. Computer assisted Pascal to Ada translation. In *Empirical Foundations of Information and Software Science IV: Empirical Methods of Evaluation of Man- Machine Interfaces. Proceedings of the Forth Symposium*, 1987.

[Per88] M. et al. Persian. Ada translation tools development: Automatic translation of Fortran to Ada. *Ada Letters*, 8(6), 1988.

[PJ92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, July 92.

[PJHH+93] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 93.

[PJL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming & Computer Architecture*, Sept 91.

[Pla89] P. J. Plauger. Translating Pascal to C. *Journal of C Language Translation*, 1(2), September 1989.

[Pla93] Eekelen Van Plasmeijer. *Functional programming and Parallel graph rewriting*. Prentice Hall, 1993.

[Pop94] Robin Popplestone. Specifying types by grammars: A polymorphic static type checker operating at a stack machine level. Technical Report FP-94-01, Department of Computer Science, University of Glasgow (Visitor from Computer Science Department, University of Massachusetts at Amherst), 94.

[Pri92] A. Prinz. Modula-2-C-compiler, release 1.1. Technical report, Humboldt-Universität zu Berlin - Institut für Informatik, October 1992.

[QRW95] P. Quinton, S. Rajopadhye, and D. Wilde. Deriving imperative code from functional programs. In *ACM SIGPLAN Intenational Conference on Functional Programming*, 95.

[RE91] Jonathan Rees and William Clinger (Editors). Revised^4 report on the algorithmic language scheme, November 1991.

[Reh95] Jakob Rehof. Polymorphic dynamic typing. aspects of inference and proof theory. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, Denmark, August 1995.

[Rev88] György Revesz. *Lambda-calculus, combinators and functional programming*. Cambridge University Press, 1988.

[RF95] Douence Rémi and Pascal Fradet. Towards a taxonomy of functional language implementations. In *PLILP;95*, LNCS. Springer Verlag, sept 95.

[Rie91] Jon G. Riecke. Fully abstract translations between functional languages (preliminary report). In *Principles of Programming Languages*, pages 245–254. ACM Press, 1991.

[SA95] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ML. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, 1995.

[Sab94] Amr Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms.* PhD thesis, Rice University, August 1994.

[San95] André Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Glasgow University, Department of Computing Science, 1995.

[Sch24] M. Schönfinkel. *Über die Bausteine der Mathematischen Logik.* Mathematische Annalen 92, 305, 1924.

[Sch92] Franz Josef Schmitt. *Praxis des Compilerbaus.* Carl Hanser Verlag, 1992.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Functional Programming & Computer Architecture*, pages 233–242, Copenhagen, June 93.

[Sho76] Edward H. Shortliffe. *Computer-Based Medical Consultation: MYCIN.* American Elsevier, 1976.

[Sin90] Duncan C. Sinclair. Solid modelling in Haskell. In *Glasgow functional programming workshop*, pages 246–263. Springer-Verlag, 1990.

[Ste93] Susan Stepney. *High Integrity Compilation.* Prentice Hall, 1993.

[Ste94] Paul Steckler. *Correct Higher-Order Program Transformations.* PhD thesis, College of Computer Science, Northeastern University, 94.

[STO+58] J. Wegstein Strong, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The problem of programming communication with changing machines. *Comunications of the ACM 1(8):12-18*, August 1958. Part 2: 1:9 9-15.

[TAL90] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, Nov 90.

[Tam93] Tanel Tammnet. Hobbit (Scheme to C compiler). Technical report, Department of Computing Science, Chalmers University of Technology, University of Goteborg S-41296, Göteborg, Sweden, 1993.

[Thi94] Peter Thiemann. *Grundlagen der funktionalen Programmierung.* B.G. Teubner Stuttgart, 1994.

[Tho85] Johnsson Thomas. Lambda Lifting: Transforming Programs to Recursive Equations. In *Conference on Functional Programming Languages and Computer Architecture, Nancy,France.* LNCS 201, 1985.

[Tho91] Simon Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.

[Tho93] Stephen Thomas. *The Pragmatics of Closure Reduction.* PhD thesis, University of Kent, Computer Science Department, 1993.

[Tho95] Simon J. Thompson. *Miranda: The Craft of Functional Programming.* Addison-Welsey, 1995.

[TL92] Simon Thompson and Rafael D Lins. The categorical multi-combinator machine: CMCM. Technical Report 20-92, Computing Laboratory, University of Kent, Canterbury, UK, July 92.

[TP86] H-C Tu and A. J. Perlis. FAC: A functional APL language. In *IEEE Software 3, 1*, pages 36–45, 1986.

[Tra89] Kenneth R. Traub. A new approach to compiling non-strict functional languages, 1989.

[Tur76] D. A. Turner. SASL language manual. Technical report, Univ. St. Andrews, 1976.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Softw. Pract. Exper.9*, pages 31–49, 1979.

[Tur81] D. A. Turner. The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on functional programming languages and computer architekture*, pages 85–92. ACM, 1981.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional programming languages and computer architekture*, pages 1–16. Springer-Verlag LNCS 201, 1985.

[TWM95] David N Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Functional Programming & Computer Architecture*, San Diego, California, 1995.

[Vol94] Kris De Volder. Continuation-Passing-Style as an Intermediate Representation for Compiling Scheme. In *GRONICS-94*, 1994.

[Wad71] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus, Chapter 4.* PhD thesis, Oxford, 1971.

[Wad92] P. Wadler. The essence of functional programming. In *XIX Symp. on Principles of Programming Languages (Santa Fe, New Mexico), Invited talk*, 1992.

[Wad95] Philip Wadler. How to declare an imperative (invited paper). In John Lloyd, editor, *ILPS'95: International Logic Programming Symposium.* MIT Press, December 1995.

[Wat89] Des Watson. *High-Level Languages and Their Compilers.* Addison Wesley, 1989.

[Wat90] David A. Watt. *Programming Language Concepts and Paradigms.* Prentice Hall, 1990.

[WC86] M. H. Williams and G. Chen. Translating Pascal for execution on a Prolog-based machine. *The Computer Journal*, 29(3), 1986.

[WC93] Leslie B. Wilson and Robert G. Clark. *Comparative Programming Languages.* Addison-Wesley, 1993.

[WC94] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. In *Conference on Lisp and Functional programming*, 1994.

[Wik92] Claes Wikstroem. Processing asn.1 specifications in a declarative language. In *Software Engineering for Telecommunication Switching Systems*, April 92.

[Wir85] Niklaus Wirth. *Programmieren in Modula-2.* Springer-Verlag, 1985.

[Wir96] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus.* Addison-Wesley, 1996.

[WK84] Sholom M. Weiss and Casimar A. Kulikowski. *A Practical Guide to Designing Expert Systems.* Rowman & Allanheld, 1984.

[WR95] Malcolm Wallace and Colin Runciman. Extending a functional programming system for embedded applications. *Software Practice & Experience*, 25(1):73–96, January 1995.

[YU93] Department of Computer Science Yale University. The Yale Haskell Compiler (y2.0). Technical report, Yale University, New Haven, CT 06520 UNITED STATES, 1993.

Anhang A

Haskell-Code für den *FQ→QTTZ*-Translator

```
--                        Macos, Dragan                    --
--                     Humboldt Universitaet zu Berlin     --
--                  FQ->QTTZ-Translator ohne Typinferenz   --


type Name         = String


type BindsType    = [(Term, Term)]

data Term         = Var Name
                  | Con Int
                  | Add Term Term
                  | Sub Term Term
                  | Mul Term Term
                  | Div Term Term
                  | Le  Term Term
                  | Lam [Term] Term
                  | App Term [Term]
                  | Let  BindsType Term
                  | If  Term Term Term
                  | IF | THEN | ELSE | ENDIF
                  | Arr  Term Term
                  | ASS
                  | APP|ARG|ENDAPP
                  | BLOCK
```

```
                | RET
                | PROC | ARGS | VARS | HEXPRS|HPROCS|HTRANSF| ENDPROC
                | BEGIN
                | ADD | SUB|MUL|DIV|ARR
                | BOTT


data Value      = Wrong
                | Num Int
                | Fun (Value ->  Value)
                | BoolType Bool


ops             = [ADD,SUB,MUL,DIV]


showval                 :: Value -> String
showval Wrong           = "<wrong>"
showval (Num i)         = show i
showval (BoolType c)    = show c
showval (Fun f)         = "<function>"

showterm                 :: Term -> String
showterm (Var name) =  "Var " ++ name
showterm (Con int)  =  " Con " ++ (show int)
showterm (Add term1 term2) = " Add " ++ (showterm term1)
                                     ++ (showterm term2)
showterm (Add term1 term2) = " Add " ++ (showterm term1)
                                     ++ (showterm term2)
showterm (Sub term1 term2) = " Sub " ++ (showterm term1)
                                     ++ (showterm term2)
showterm (Mul term1 term2) = " Mul " ++ (showterm term1)
                                     ++ (showterm term2)
showterm (Div term1 term2) = " Div " ++ (showterm term1)
                                     ++ (showterm term2)
showterm (Le term1 term2) = " Le " ++ (showterm term1)
                                   ++ (showterm term2)
showterm (App term1 term2)  = " App " ++ (showterm term1)
                                      ++ (showterms term2)
showterm (Arr term1 term2)     = " Arr " ++ (showterm term1)
                                         ++ (showterm term2)
```

```
showterm (If cond thenopt elseopt)  =  "If " ++ " "
                                       ++(showterm cond)
                                       ++ " "
                                       ++ (showterm thenopt)
                                       ++ (showterm elseopt)
showterm (Let binds expr) = "Let " ++ (show (showpairterms binds))
                                   ++ "IN "
                                   ++(showterm expr)
                            where showpairterms [] = []
                                  showpairterms ((a,b):r) =
                                  ((showterm a), (showterm b)):
                                  (showpairterms r)

showterm IF =   "  IF  "
showterm THEN =   " THEN "
showterm ELSE =   " ELSE "
showterm ASS =   " ASS "
showterm BLOCK =   " BLOCK "
showterm RET =   " RET "
showterm PROC =   " PROC "
showterm ARGS =   " ARGS "
showterm VARS =   " VARS "
showterm HEXPRS =   " HEXPRS "
showterm BEGIN =   " BEGIN "
showterm HTRANSF =   " HTRANSF "
showterm ENDPROC =   " ENDPROC "
showterm ENDIF   =  " ENDIF "
showterm HPROCS =   " HPROCS "
showterm ADD =   " ADD "
showterm SUB =   " SUB "
showterm MUL =   " MUL "
showterm DIV =   " DIV "
showterm ARR =  " ARR "
showterm ENDAPP = " ENDAPP "
showterm APP = " APP "
showterm ARG = " ARG "
showterm (Lam names term)  = "Lam"++(showterms names )++" "
                                  ++(showterm term)

showterms :: [Term] -> String
showterms l = mapDM showterm l
```

```
              where mapDM f [] = []
                    mapDM f (x:xs) = (f x)++(mapDM f xs)

translator::Int->Name->[Term]->[Term]->[Term]->[Term]->
            [Term]->[Term]->[Term]->[Term]

-- Globale "Ubersetzung f"ur LET(REC)
-- Regel 1

translator s fun fp ((Let binds term):e) htransf
           hexpr hvars hprocs arr =
         translator s fun fp
                    ((putass exprlist)++(term:e))
         htransf (varslist++hexpr)
                    (varslist++hvars) hprocs arr
            where varslist = vars binds
                  exprlist = exprs binds

-- "Ubersetzung f"ur a := lambda x.e
-- Regel 2

translator s fun fp (ASS:Lam formals term:e) htransf
           ((Var var) : hexpr) hvars hprocs arr =
  translator s fun fp e
             htransf hexpr (takeout (Var var) hvars)
             (translator s var  formals (term:[])  [] [] [] []
             ((Arr (Var var) (Con (length formals))) :arr))
             ((Arr (Var var) (Con (length formals))) :arr)

-- "Ubersetzung von a := let...
-- Regel 3

translator s fun fp (ASS: (Let binds term):e) htransf
           hexpr hvars hprocs arr =
               translator s fun fp
                          ((putass exprlist)++(ASS:term:e))
                          htransf (varslist++hexpr) (varslist++hvars)
                          hprocs arr
               where varslist = vars binds
                     exprlist = exprs binds
```

```
-- "Ubersetzung von a = if then else
-- Regel 4

translator s fun fp (ASS:If cond thenopt elseopt:e)
           htransf (var: hexpr) hvars hprocs arr =
             translator s fun fp
               (IF:cond:THEN:ASS:thenopt:ELSE:ASS:elseopt:ENDIF:e)
               htransf (var:var:hexpr) hvars hprocs arr

-- "Ubersetzung von IF
-- Regel 5

translator s fun fp (If cond thenopt elseopt:e) htransf hexpr hvars
           hprocs arr =
               translator s fun fp
                    (IF:cond:THEN:thenopt:ELSE:elseopt:ENDIF:e)
                    htransf hexpr hvars hprocs arr

-- "Ubersetzung von Ass App
-- Regel  6

translator s fun fp (ASS:(App f args):e)  htransf (a:hexpr)
           hvars hprocs arr =
              if  (iscurry (App f args) arr)
              then translator (s+(arrity f arr)) fun fp
              ((putass args)++(ASS:[Lam helpvarsNK
               (App f (helpvarsK++helpvarsNK))]++e))
               htransf (helpvarsK++(a:hexpr))
                       (helpvarsK++hvars) hprocs
               ((Arr a (Con (length helpvarsK ))):arr )
    else translator s fun fp ((App f args):e)
         (a:ASS:htransf) hexpr hvars hprocs arr
    where { helpvarsK  =  newVars s (length args);
            helpvarsNK = newVars (s+(length args)+1)
                                 ((arrity f arr)-(length args))}

-- "Ubersetzung von App
-- Regel 7

translator s fun fp ((App f args):e)  htransf
           hexpr hvars hprocs arr =
```

```
  translator s fun fp ((APP:f:(putterm ARG args))++(ENDAPP:e))
             htransf hexpr hvars hprocs arr

-- ASS
-- Regel 8

translator s fun fp (ASS:e:eglob) htransf (a:hexpr)
           hvars hprocs arr =
  translator s fun fp (e:eglob) (a:ASS:htransf)
             hexpr hvars hprocs arr

-- Sprachkonstrukte, die keine zus"atzlichen Transformationen verlangen
-- Regel 9

translator s fun fp (schw:e) htransf hexpr hvars hprocs arr =
  translator s fun fp e (schw:htransf) hexpr hvars hprocs arr

-- Abschliessende Erzeugung des QTTZ-Zielcodes
-- Regel 10

translator s fun fp [] htransf hexpr hvars hprocs arr =
 ((PROC:Var fun:ARGS:fp)++((VARS:hvars)++(HPROCS :hprocs)++
  (BEGIN:(reverse (htransf)))++ ([ENDPROC])++(ARR:arr)))

exprs :: BindsType -> [Term]
exprs [] = []
exprs ((name,term):r) = (term:(exprs r))

vars :: BindsType -> [Term]
vars [] = []
vars ((name,term):r) = (name:(vars r))

putterm ::Term->[Term]->[Term]
putterm _ []     = []
putterm t (x:xs) = t:x:(putterm t xs)

putass :: [Term] -> [Term]
putass           = putterm ASS

instance Eq Term where
 Var name1 == Var name2 = name1 == name2
```

```
 Con int1  == Con int2  = int1  == int2
 _         == _         = False


member ::  (Eq a)  => a->[a]->Bool
member x [] = False
member x (y:ys) =  if x==y  then True
                            else member x ys

takeout ::   Term ->[Term]->[Term]
takeout x [] = []
takeout x (y:xs) = if x==y  then takeout x xs
                            else (y:(takeout x xs))

makeset :: (Eq a) => [a]->[a]
makeset []     = []
makeset (x:xs) = if (member x xs)
                 then makeset xs
                 else (x:(makeset xs))

newVars s br  | br == 0 = []
              | otherwise = (Var ("$" ++(show s))) :
                            (newVars (s+1) (br-1))

-- curry :: Term->[Term]->Bool
iscurry (App f args) arr = not ((length args) == (arrity f arr))

arrity  :: Term->[Term]->Int
arrity f [] = 0
arrity f ((Arr (Var name) (Con length)):xs) =
                      if   f == Var name
      then length
  else arrity f xs

term0 = (App
           (App (Lam [Var "x"] (Lam [Var "y"]
                                (Add (Var "x") (Var "y")))
                )
                [(Con 10)]
           )
           [(Con 11)]
```

```
        )

term4 = (If (App (Var "a") [(Var "x")] ) (Con 1)
                                    (App (Var "a") [(Con 20)]))

test  = showterms (translator 0 "Prog" [] [term1test] [] [] [] []
                   arritylist)

test1 = showterms (translator 0 "main" [] [term1test] [] [] []
                   [] [])

term1test = Let [(Var "a", (Lam [Var "x"] (If
                                          (App (Var "<")
                                               [(Var "x"),
                                                (Con 1)
                                               ]
)
(Var "x")
(Con 1))))]
                               (App (Var "a") [(Con 1)])

test2 = showterms (translator 0 "main" [] [term2test] [] [] [] []
                   arritylist)

term2test = Let [(Var "a", (Lam [Var "x"] (Var "x"))),
                            (Var "b", Con 2),
                            (Var "c", Con 3)]
                       (If (Var "b") (Var "c")
                           (App (Var "a") [Con 10]))

test4 = showterms (translator 0 "main" [] [term4test] [] [] [] []
                   arritylist)

term4test = Let [(Var "a", Con 1), (Var "b",
                                     (If (Var "a") (Con 1) (Con 2)))]
             (Let [(Var "c", Con 2)] (Var "c"))

test5 = showterms (translator 0 "main" [] [term5test] [] [] [] []
                   arritylist)

term5test = App (Var "+") [(Con 3),(Con 4)]
```

```
test6 = showterms (translator 0 "main" [] [term6test] [] [] [] []
                   arritylist)

term6test = App (App (Var  "+") [(Con 3)]) [Con 4]

term7test=  Let [(Var "a", ( App (Var  "+")
 [(Con 1)]))]
 (App (Var "a") [(Con 2)])

test7 = showterms (translator 0 "main" [] [term7test] [] [] [] []
                   arritylist)

term3 = ( If (Le (Con 3)  (Con 2)) (Con 10) (Con 20))

arritylist = [Arr (Var "+") (Con 2), Arr (Var "-") (Con 2),
              Arr (Var "*")  (Con 2), Arr (Var "<") (Con 2),
              Arr (Var ">") (Con 2)]
```

Anhang B

Beispiele der Übersetzung einer funktionalen in eine prozedurale Sprache mit dem *SMC*

Symbolische Differentiation

```
(LETREC
  ((DIFF1
      (LAMBDA (E)
         (IF (not (list? E))
           (IF (equal? E (QUOTE X))
                1
                0
           )
           (IF (equal? (CAR E) (QUOTE PLUS))
               (LET ( (E1 (CAR (CDR E)))
                      (E2 (CAR (CDR (CDR E))))
                    )
                    (SUM (DIFF1 E1) (DIFF1 E2)))
               (IF (equal? (CAR E) (QUOTE MAL))
                   (LET ((E1 (CAR (CDR E)))
                         (E2 (CAR (CDR (CDR E))))
                        )
                        (SUM (MULTIPL E1 (DIFF1 E2))
                             (MULTIPL (DIFF1 E1) E2)))
                    (QUOTE Error))))))

    (SUM
      (LAMBDA (U V)
         (CONS (QUOTE PLUS)
               (CONS U (CONS V (QUOTE ()))))))
     (MULTIPL
      (LAMBDA (U V)
         (CONS (QUOTE MAL)
               (CONS U (CONS V (QUOTE ())))))))
 (DIFF1 (QUOTE (PLUS (MAL x x) 2)))
)
```

```
MODULE diff ;

FROM SExps IMPORT
  QuoteSE,  DispSE, EqualSE, EqSE, HeadSE,
  TailSE, ConsSE,  QIntSE, IsTypeSE, InitSE, NullSE;
FROM SExpsKr3 IMPORT
  SExp, ConstrSE;
```

```
FROM InOut IMPORT
  WriteInt ;

PROCEDURE DIFF1 (E : SExp):SExp ;
BEGIN
  IF NOT IsTypeSE(E, ConstrSE) THEN
    IF ( EqualSE ( E, QuoteSE ( 'X' ) ) ) THEN
      RETURN QIntSE(1)
    ELSE
      RETURN QIntSE(0)
    END ;
  ELSE
    IF (EqualSE(HeadSE(E), QuoteSE('PLUS'))) THEN
      RETURN SUM(DIFF1(HeadSE(TailSE(E))), DIFF1(HeadSE(TailSE(TailSE(E)))))
    ELSE
      IF (EqualSE(HeadSE(E), QuoteSE('MAL'))) THEN
        RETURN SUM(MULTIPL(HeadSE(TailSE (E)),
                           DIFF1(HeadSE(TailSE(TailSE(E))))),
                   MULTIPL (DIFF1(HeadSE(TailSE(E))),
                            HeadSE(TailSE(TailSE(E)))))
      ELSE
        RETURN QuoteSE('ERROR')
      END ;
    END ;
  END ;
END DIFF1 ;

PROCEDURE SUM (U : SExp;
               V : SExp) : SExp;
BEGIN
  RETURN ConsSE(QuoteSE('PLUS'), ConsSE(U, ConsSE(V, NullSE)))
END SUM ;

PROCEDURE MULTIPL (U : SExp;
                   V : SExp) : SExp;
BEGIN
  RETURN ConsSE(QuoteSE('MAL'), ConsSE(U, ConsSE(V, NullSE)))
END MULTIPL ;

BEGIN
  InitSE ;
  DispSE(DIFF1(QuoteSE('(PLUS (MAL X X) 2)')))
END diff.
```

Fibonacci-Zahlen

```
(letrec
  ((fib
      (lambda (n)
        (if (or (= n 1)  (= n 0))
           1
           (+ (fib (- n 1)) (fib (- n 2)))))))

  (fib 30))
```

```
MODULE fib1eq;

FROM InOut IMPORT
  WriteInt;

PROCEDURE FIB(N : INTEGER) : INTEGER;

BEGIN
  IF ((N = 1) OR (N = 0)) THEN
    RETURN 1
  ELSE
   RETURN (FIB(N - 1) + FIB( N - 2))
  END;
END FIB;


BEGIN
 WriteInt(FIB(30), 4)
END fib1eq.
```

Fakultätfunktion

```
( letrec
 ((fakt
   (lambda (n)
     (if (= n  1)
       1
       (* n (fakt (- n  1)))))))

 (fakt 10))
```

```
MODULE fakt1;

FROM InOut IMPORT
  WriteInt;

PROCEDURE FAKT(N : INTEGER) : INTEGER;
BEGIN
  IF (N = 1) THEN
    RETURN 1
  ELSE
    RETURN (N * FAKT((N - 1)))
  END ;
END FAKT ;

BEGIN
  WriteInt(FAKT(10), 4)
END fakt1.
```

Problem der n Damen

```
( LETREC
 ((QUEEN
     (LAMBDA (N)
        (FIND 1 1 (QUOTE ()) N)))

   (FIND
      (LAMBDA (I J BOARD N)
         (IF (PROBLEM I J BOARD)
            (ADVANCE I J BOARD N)
            (IF  (< N (+ I   1))
               (REVERSE (CONS (LIST I J) BOARD))
               (FIND (+ I 1) 1 (CONS (LIST I J) BOARD) N)))))
   (PROBLEM
      (LAMBDA (I J BOARD)
        (IF  (EQUAL? BOARD  (QUOTE ()))
           #F
           (OR (THREATEN I J (CAR (CAR BOARD)) (CAR (CDR (CAR BOARD))))
               (PROBLEM I J (CDR BOARD))))))

   (THREATEN
      (LAMBDA (I J A B)
         (OR (OR (OR (EQ? I A) (EQ? J B)) (EQ? (- I J) (- A B)))
             (EQ? (+ I J) (+ A B)))))

   (ADVANCE
      (LAMBDA (I J BOARD N)
         (IF  (< N (+ J   1))
            (REMOVE  BOARD N)
            (FIND I (+ J  1) BOARD N))))

   (REMOVE
     (LAMBDA ( BOARD N)
        (IF  (equal? BOARD  (QUOTE ()))
           (QUOTE (IMPOSIBBLE))
           (ADVANCE (CAR (CAR BOARD))
                    (CAR (CDR (CAR BOARD)))
                    (CDR BOARD) N))))
   (REVERSE
      (LAMBDA (L)
         (REV L  (QUOTE ())))))
```

```
   (LIST
     (LAMBDA (L1 L2)
        (CONS L1 (CONS L2 (QUOTE ()))))))

   (REV
      (LAMBDA (L1 L2)
         (IF  (equal? L1 (quote ()))
            L2
            (REV (CDR L1) (CONS (CAR L1) L2))))))

   (QUEEN 8)
)
```

```
MODULE queen ;

FROM SExps IMPORT
  QuoteSE, DispSE, EqualSE, EqSE,
  HeadSE, TailSE, ConsSE, QIntSE, ValIntSE, IsNullSE,  InitSE;

FROM SExpsKr3 IMPORT
  SExp ;


PROCEDURE QUEEN1(N : INTEGER) : SExp;
BEGIN
  RETURN FIND(QIntSE(1), QIntSE(1), QuoteSE('NIL'), N)
END QUEEN1 ;

PROCEDURE FIND(I : SExp;
               J : SExp;
               BOARD : SExp;
                 N : INTEGER) : SExp;
BEGIN
  IF PROBLEM(ValIntSE(I), ValIntSE(J), BOARD) THEN
    RETURN ADVANCE(I, ValIntSE(J), BOARD, N)
  ELSE
    IF (N < (ValIntSE(I) + 1)) THEN
      RETURN REVERSE(ConsSE(LIST(I, J), BOARD))
    ELSE
      RETURN FIND(QIntSE((ValIntSE(I) + 1)),
                  QIntSE(1), ConsSE(LIST(I, J), BOARD), N)
```

```
    END;
  END;
END FIND;

PROCEDURE PROBLEM(I : INTEGER;
                  J : INTEGER;
                  BOARD : SExp) : BOOLEAN;
BEGIN
  IF(EqualSE(BOARD, QuoteSE('NIL'))) THEN
    RETURN FALSE
  ELSE
    RETURN(THREATEN(I, J, ValIntSE(HeadSE(HeadSE(BOARD))),
             ValIntSE(HeadSE(TailSE(HeadSE(BOARD))))) OR
             PROBLEM(I, J, TailSE(BOARD)))
  END ;
END PROBLEM ;

PROCEDURE THREATEN ( I : INTEGER;
                     J : INTEGER;
                     A : INTEGER;
                     B : INTEGER) : BOOLEAN;
BEGIN
  RETURN((((I = A) OR (J = B))
         OR ((I - J) = (A - B)))
         OR ((I + J) =  (A + B)))
END THREATEN ;

PROCEDURE ADVANCE (I : SExp;
                   J : INTEGER;
                   BOARD : SExp;
                   N : INTEGER) : SExp;
BEGIN
  IF (N <  (J + 1)) THEN
    RETURN REMOVE(BOARD, N)
  ELSE
    RETURN FIND( I, QIntSE((J + 1)), BOARD, N)
  END;
END ADVANCE;

PROCEDURE REMOVE(BOARD : SExp;
                 N : INTEGER) : SExp;
BEGIN
  IF (EqualSE(BOARD, QuoteSE('NIL'))) THEN
    RETURN QuoteSE('IMPOSIBBLE')
```

```
  ELSE
    RETURN ADVANCE(HeadSE(HeadSE(BOARD)),
           ValIntSE(HeadSE(TailSE(HeadSE(BOARD)))),
           TailSE(BOARD), N)
  END;
END REMOVE;

PROCEDURE REVERSE(L : SExp) : SExp;
BEGIN
  RETURN REV(L, QuoteSE('NIL'))
END REVERSE;

PROCEDURE REV(L1 : SExp;
              L2 : SExp) : SExp;
BEGIN
  IF (EqualSE(L1, QuoteSE('NIL'))) THEN
    RETURN L2
  ELSE
    RETURN REV(TailSE(L1), ConsSE(HeadSE(L1), L2))
  END;
END REV;

PROCEDURE LIST(X : SExp;
               Y : SExp) : SExp;
BEGIN
  RETURN ConsSE (X, ConsSE(Y, QuoteSE('NIL')))
END LIST;

BEGIN
  InitSE;
  DispSE(QUEEN1(8))
END queen.
```

Funktion *Tak*

```
(LETREC
 ((TAK
   (LAMBDA (X Y Z)
      (IF (not (< Y X))
        Z
        (TAK (TAK (- X  1) Y Z)
             (TAK (- Y  1) Z X)
             (TAK (- Z 1) X Y))))))
 (TAK 11 7 50)
)
```

```
MODULE tak;
FROM InOut IMPORT
  WriteInt;


PROCEDURE TAK ( X : INTEGER;
                Y : INTEGER;
                Z : INTEGER) : INTEGER ;
BEGIN
  IF NOT (Y < X) THEN
    RETURN Z
  ELSE
    RETURN TAK(TAK((X - 1), Y, Z), TAK((Y - 1), Z, X), TAK((Z - 1), X, Y))
  END;
END TAK;

BEGIN
  WriteInt(TAK(11, 7, 50), 4)
END tak.
```

Programm für die Berechnung der Vereinigungsmenge der Unterlisten einer Liste

```
(LETREC ((UNION1( LAMBDA (LS)
               (IF (NULL? LS)
                   (QUOTE ())
                   (UNION2 (CAR LS) (UNION1 (CDR LS))))))

          (UNION2 (LAMBDA (S1 S2)
               (MAKESET (APPEND S1 S2))))

          (MAKESET (LAMBDA (S)
               (IF (NULL? S)
                  (QUOTE ())
                  (CONS (CAR S) (MAKESET (TAKEOUT (CAR S) (CDR S)))))))

          (TAKEOUT (LAMBDA (X S)
                (IF (NULL? S)
                   (QUOTE () )
                   (IF (EQUAL? X (CAR S))
                      (TAKEOUT X (CDR S))
                      (CONS (CAR S) (TAKEOUT X (CDR S))))))))

        (UNION1 (QUOTE ((1 2 3 23 23 23 2 2 3 345 4 45 4  45 4 5 45 4 5  4 5 45)
                        ( 5 6) ( 2 3))))
)
```

```
MODULE union;

FROM SExps IMPORT
  QuoteSE, DispSE, HeadSE, TailSE, ConsSE,
  QIntSE, InitSE, EqualSE, IsNullSE, NullSE;
FROM SExpsKr3 IMPORT
  SExp;

FROM SELib IMPORT
  AppSE;
```

```
PROCEDURE UNION1(LS : SExp) : SExp;
BEGIN
  IF IsNullSE(LS) THEN
    RETURN NullSE
  ELSE
    RETURN UNION2(HeadSE(LS), UNION1(TailSE(LS)))
  END;
END UNION1;

PROCEDURE UNION2(S1 : SExp;
                 S2 : SExp) : SExp;
BEGIN
  RETURN MAKESET(AppSE(S1, S2))
END UNION2;

PROCEDURE MAKESET(S : SExp) : SExp;
BEGIN
  IF NULL(S) THEN
    RETURN NullSE
  ELSE
    RETURN ConsSE(HeadSE(S), MAKESET(TAKEOUT(HeadSE(S), TailSE(S))))
  END;
END MAKESET;

PROCEDURE TAKEOUT(X : SExp;
                  S : SExp) : SExp;
BEGIN
  IF NULL(S) THEN
    RETURN NullSE
  ELSE
    IF (EqualSE(X, HeadSE(S))) THEN
      RETURN TAKEOUT(X, TailSE(S))
    ELSE
      RETURN ConsSE(HeadSE(S), TAKEOUT(X, TailSE(S)))
    END;
  END;
END TAKEOUT;

PROCEDURE NULL(L:SExp) : BOOLEAN;
BEGIN
  RETURN (EqualSE (L, NullSE))
END NULL;
BEGIN
```

```
  InitSE ;
  DispSE (UNION1 (QuoteSE('((1 2 3 23 23 23 2 2 3 345 4 45 4 45 4 5
                            45 4 5 4 5 45)
                           (5 6) (2 3))')))
END union.
```

Erklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Berlin, den 15.01.97

Lebenslauf

18.04.67	geboren in Zrenjanin, Jugoslawien
09/82-06/84	Gymnasium (Mittelstufe) in Novi Becej, Jugoslawien
09/84-05/86	Gymnasium (Oberstufe), Fachrichtung Informatik
05/86	Abitur, Gesamtnote: 5.00 (5 ist die beste Note)
09/86-09/87	Wehrdienst
10/87-07/93	Studium an der Universität in Novi Sad, Fakultät der Naturwissenschaften und Mathematik
07/93	Abschluß: „Diplom -Mathematiker"
04/93-10/93	Wissenschaftlicher Mitarbeiter an der Fakultät der Naturwissenschaften und Mathematik der Universität in Novi Sad
04/94-07/97	Doktorand an der Lehr- und Forschungseinheit „Softwaretechnik und Theorie der Programmierung"
05/97-09/97	Freier Mitarbeiter bei Langmack&Partner - „Softlab" (BMW): Softwareentwicklung, Beratung im Gebiet Compilerbau
seit 09/97	Wissenschaftlicher Mitarbeiter an der Lehr- und Forschungseinheit „Softwaretechnik und Theorie der Programmierung" der Humboldt-Universität zu Berlin